

QA
276.4
B42
1981
C.2

C.S.

S
A Language and System
for Data Analysis

Richard A. Becker
John M. Chambers



Bell Laboratories
January, 1981

Table of Contents

Section	Page
1. Introduction to the S User's Guide.	1-1
1.1. Outline; Suggested Approach.	1-1
1.2. Concepts in S: Expressions and Data.	1-1
1.3. Invoking S.	1-2
2. Basic Use of S.	2-1
2.1. The Language.	2-1
2.1.1. Expressions; Operators; Functions.	2-1
2.1.2. Interacting with S: Errors; Interrupts; Help.	2-3
2.2. Data.	2-5
2.2.1. Data Values.	2-5
2.2.2. Vectors.	2-5
2.2.3. Data Input.	2-6
2.2.4. Databases.	2-7
2.2.5. Data Structures: Matrices and Time-Series.	2-8
2.3. Data Analysis.	2-11
2.3.1. Statistical Functions.	2-11
2.3.2. Data Manipulation.	2-13
2.3.3. Numerical Calculations.	2-15
2.4. Graphics.	2-16
2.5. Example.	2-19
3. Advanced Use of S.	3-1
3.1. Language.	3-1
3.1.1. Extending the Language: Source and Sink; Macros.	3-1
3.1.2. Applying Functions; Iteration.	3-3
3.1.3. Conditional expressions; compound expressions.	3-6
3.1.4. More on Arithmetic and Operators.	3-7
3.1.5. Numbered Components of a Structure.	3-8
3.1.6. Executing Commands on Invoking S.	3-8
3.1.7. Batch Execution of S.	3-9
3.1.8. Keeping Track of an Analysis: Diary.	3-9
3.1.9. Functional Form of Operations	3-10
3.2. Data.	3-11
3.2.1. Structures: Arrays; Vector Structures.	3-11
3.2.2. Prefixes.	3-12
3.2.3. Attaching Databases; Search Lists.	3-13
3.2.4. Moving Data between Computers; Nonstandard Input.	3-14
3.2.5. NULL Data Type.	3-16
3.2.6. Categorical Variables.	3-16
3.2.7. Documenting Datasets and Macros.	3-17
3.3. Data Analysis.	3-18
3.3.1. Statistical Functions.	3-18
3.3.1.1. More on Regression and Models.	3-18

3.3.1.2.	Probability and Quantile Functions.	3-20
3.3.1.3.	Multivariate Analysis.	3-20
3.3.1.4.	Matrix Methods; Linear Algebra.	3-22
3.3.2.	Data Manipulation.	3-22
3.4.	Graphics.	3-23
3.4.1.	More High-Level Functions.	3-23
3.4.2.	Contour and Perspective Plots of Surfaces.	3-26
3.4.3.	Multivariate Plotting Functions.	3-27
3.4.4.	Graphical Data Structures; Missing Values.	3-29
3.4.5.	Graphical Input; Identify, Rdpn.	3-30
3.4.6.	Graphical Parameters.	3-30
3.4.7.	The Layout of Plots and Figures.	3-33
3.4.8.	Building-up Plots.	3-35
3.4.9.	Deferred Graphics.	3-38
4.	The S Macro Processor.	4-1
4.1.	Macros in S.	4-1
4.2.	Defining Macros.	4-2
4.3.	Literals; Conditional Expansion.	4-5
4.4.	Macros with Many Arguments.	4-7
4.5.	User-Oriented Macros.	4-9
4.6.	Temporary Definitions; Recursive Calls.	4-11
4.7.	Built-In Macros and Special Constructions.	4-12
4.8.	Examples of S Macros.	4-13
5.	Reference.	5-1
5.1.	The Language: Syntax.	5-1
5.1.1.	Function Calls and Commands.	5-3
5.1.2.	Compound Expressions.	5-4
5.1.3.	Continuation.	5-4
5.1.4.	Reserved words.	5-4
5.2.	Semantics.	5-5
5.2.1.	Functions and Operators.	5-5
5.2.2.	Side Effects: Database Changes and Parameters.	5-7
5.2.3.	Compound Expressions.	5-8
5.2.4.	Conditional Expressions; Iterative Expressions.	5-8
5.3.	Data Structures.	5-9
6.	Writing New Functions for S.	6-1
6.1.	Design and Implementation of Simple S Functions.	6-2
6.1.1.	Design of S Functions.	6-2
6.1.2.	Arguments; the FUNCTION Statement.	6-2
6.1.3.	Error Checking.	6-3
6.1.4.	Dynamic and Static Data Structures.	6-4
6.1.5.	Computations.	6-5
6.1.6.	Creating New Functions.	6-6
6.1.7.	Hints; Debugging.	6-8
6.2.	Data Structures.	6-10
6.2.1.	Data Values and Attributes.	6-10
6.2.2.	Character Data and Character Attributes.	6-10

6.2.3.	Missing Values.	6-12
6.2.4.	Structures and Components.	6-13
6.2.5.	Modes Computed at Execution.	6-13
6.3.	Function Arguments.	6-17
6.3.1.	Arguments in the FUNCTION Statement.	6-17
6.3.2.	Interrupting and Resuming Argument Processing.	6-18
6.3.3.	Arbitrarily Many Arguments.	6-19
6.3.4.	Treating Structures Like Argument Lists.	6-20
6.4.	Function Results and Related Statements.	6-21
6.4.1.	The RETURN Statement.	6-22
6.4.2.	CHAIN: Invoking Another Function.	6-23
6.4.3.	INSERT: Building Structures.	6-23
6.5.	Graphics Functions.	6-24
6.5.1.	Declaring a Graphics Function.	6-24
6.5.2.	Graphical Parameters.	6-24
6.5.3.	Plotting Data Structure.	6-25
6.5.4.	High Level Graphics Functions: SETUP and LOGPLOT.	6-25
7.	Writing and Using Algorithms	7-1
7.1.	The Algorithm Language: Basics.	7-1
7.1.1.	Languages.	7-1
7.1.2.	MAKE: Generating S Functions and Stand-Alone Programs.	7-1
7.1.3.	Error Handling.	7-2
7.1.4.	Symbolic Constants; Declarations.	7-3
7.1.5.	Debugging.	7-3
7.1.6.	C Language Facilities.	7-4
7.2.	Special Facilities.	7-4
7.2.1.	Printing and Encoding.	7-4
7.2.1.1.	Basic Message Printing.	7-4
7.2.1.2.	Encoding.	7-6
7.2.1.3.	Detailed Format Control.	7-7
7.2.1.4.	Problems with Encoding.	7-8
7.2.2.	Reading and Decoding.	7-8
7.2.2.1.	Basic Reading of Data Items.	7-9
7.2.2.2.	Line Input; End-of-File.	7-10
7.2.3.	File Access; Standard Files.	7-10
7.2.4.	Dynamic Storage.	7-11
7.2.5.	Data Structures for S.	7-12
7.3.	Available Algorithms.	7-14
7.3.1.	Data Handling; Character Data.	7-14
7.3.2.	Sort and Order.	7-15
7.3.3.	Range of Data.	7-16
7.3.4.	Probabilities; Quantiles; Pseudorandom Numbers.	7-17
7.3.4.1.	Available Algorithms.	7-17
7.3.4.2.	New Pseudorandom Generators for S.	7-17
7.3.5.	Matrices and Arrays.	7-18
8.	Graphical Algorithms.	8-1
8.1.	Basic Concepts of Graphical Algorithms.	8-1
8.1.1.	Figures and Plots.	8-2

8.1.2.	User and Margin Coordinate Systems.	8-2
8.1.3.	Graphical Parameters.	8-3
8.2.	Creating Graphical Algorithms.	8-4
8.2.1.	Initialization of Graphical Algorithms.	8-4
8.2.2.	Setting-up Coordinate Systems and Axes.	8-5
8.2.3.	Drawing the Picture.	8-5
8.2.4.	Titles and Axis Labels.	8-6
8.2.5.	Specifying and Querying Graphical Parameters.	8-6
8.2.6.	Wrapping Up.	8-8
8.3.	The Structure of a Graphical Algorithm.	8-10
8.3.1.	High-Level Graphical Algorithms; S Functions.	8-11
8.3.2.	Algorithms that Augment a Plot.	8-14
8.4.	Advanced Graphical Algorithms.	8-16
8.4.1.	Control of Figures, Plots, Margins.	8-16
8.4.2.	Margins and Outer Margins.	8-18
8.4.3.	Parameters of Physical Size.	8-19
8.4.4.	Setting-up Coordinate Systems and Axes.	8-20
8.4.5.	Summary of Graphical Parameters.	8-22
8.4.6.	Graphical Input.	8-24
8.4.7.	Debugging.	8-24
8.5.	Available Graphical Subroutines.	8-24
8.6.	Stand-Alone Graphical Algorithms.	8-25
8.7.	Device Drivers.	8-26
8.7.1.	Organization of Device Driver Routines.	8-26
8.7.2.	Portability Considerations.	8-29
8.7.3.	Control of Graphic Input.	8-29
8.7.4.	Writing a Device Driver for S.	8-29

9. S Detailed Documentation.

9.1.	Function Documentation.
9.2.	Topic - Documentation Index.
9.3.	System Macros Detailed Documentation.
9.4.	System Datasets Detailed Documentation.
9.5.	Algorithm Documentation.
9.6.	Graphical Algorithm Documentation.

10. Index.

Introduction to the S User's Guide

1. Introduction to the S User's Guide.

1.1. Outline; Suggested Approach.

This user's guide describes the S language and system for interactive data analysis. Section 2 presents basic material needed for using S. Section 3 introduces more advanced material. Section 4 describes the S Macro Processor. Section 5 is a reference document which defines the full scope of portions of the language. Section 6 describes how to write new S functions, and section 7 describes the writing and use of underlying algorithms. Graphical algorithms are described in section 8, and detailed documentation of each S function is available in section 9. An index to the user's guide is provided in section 10, and should be used when information on a specific topic is needed.

Each of sections 2 and 3 of the user's guide is divided into 4 subsections, corresponding to: Language; Data; Statistics; and Display. Sections 2.1 and 3.1 discuss the expressions in the S language. Section 3.1 also introduces techniques for extending the language through the use of macros. Sections 2.2 and 3.2 cover the organization of data into structures and the management and manipulation of these structures. Sections 2.3 and 3.3 present statistical functions and related computations for data analysis. Sections 2.4 and 3.4 discuss graphical techniques and ways of displaying data. Material in sections 5 through 8 is more advanced, and should not be needed by the casual S user.

The remainder of section 1 introduces some key concepts in S. Readers are encouraged to read through this material the first time they encounter S, before they get down to specifics. The concepts are simple and general, but grasping them from the start will lead more quickly to good use of S.

1.2. Concepts in S: Expressions and Data.

S is an *expression* language. The user types expressions which are interpreted and evaluated by the S system. The expressions include, but are not limited to, conventional algebraic and functional expressions:

```
1 + y
log( 1 + yold * weight )
p > .95
x - mean(x)
```

Expressions are composed from operators and other special characters (plus, minus, parentheses, etc.), from functions (*log* and *mean*, e.g.), and from data (constants and named datasets).

Data in S is organized into named, self-describing *datasets*. These are kept in *databases* and retrieved automatically when named in an expression. The fundamental data structure is a *vector*; that is, a sequence of data values (numbers, logical values or character strings). An unlimited variety of other data structures can be built up from vectors. Some structures are recognized by many functions in S; for example, *matrices*, *arrays* and *time-series*.

Fundamental to the philosophy of S is the uniformity of expressions and data structures. The value of any expression, however complicated, is a data structure. This includes anything from simple arithmetic to high-level statistical functions. So far

as the language is concerned, any expression can appear as an argument to an operator or function. Of course, a particular function may generate an error if it cannot interpret the argument sensibly. As we proceed, examples will show how this uniformity leads to flexible, general data analysis.

1.3. Invoking S.

The use of S requires some basic familiarity with interactive computing: how to access the computer, what keys to use on the terminal to delete characters and lines and to send interrupts, etc. This elementary training is generally available through the introductory manuals for your computer system.

Once you are able to log onto the computer, type the capital letter

S

to start running S. S will prompt with the character ">". When finished with an S session, type "q" to quit.

S is a *case sensitive* language: corresponding upper- and lower-case letters are distinct in the names of functions and datasets. Thus the name *ABC* is not the same as the name *abc*. By convention, functions and system datasets have lower case names. Special built-in values such as TRUE, FALSE, and NA are upper case.

Basic Use of S

2. Basic Use of S.

This section contains the techniques which are most often needed while using S. Readers are encouraged to read this section, familiarize themselves at least generally with the ideas in it, and then try a few simple analyses similar to those shown here.

2.1. The Language.

2.1.1. Expressions; Operators; Functions.

The S user types expressions to be interpreted and executed. The result of an expression may be to compute a vector or other data structure, to assign data a name on a database for further retrieval, to generate printed or graphical output, or to control the operation of S.

Operators in S include the usual arithmetic and logical operators. These operate on all the data values in the vectors or other operands. Thus,

$y + 1$

computes its result by adding 1 to each data value in y . The arithmetic operators are:

$+$ $-$ $*$ $/$ $**$ $^$

The operators $**$ and $^$ are alternatives for raising to a power. Two less common operations are:

$\% /$ $\% \%$

The operator $\% /$ is integer divide; i.e., it divides and then truncates the quotient to an integer. The operator $\% \%$ is a remainder or modulo operator: it gives the remainder when its first operand is divided by its second. There are logical operators for comparison of data values, which produce the values TRUE or FALSE:

$<$ $>$ $<=$ $>=$ $==$ $!=$

for less-than, greater-than, less-than-or-equal-to, greater-than-or-equal-to, equal-to and not-equal-to. There are also boolean operators

$\&$ $|$ $!$

that are used for the operations *and*, *or*, and *not*.

The S operator $:$ creates the sequence between any two single numbers in steps of 1:

> 9:23

```
      9 10 11 12 13 14 15 16 17 18
[ 11] 19 20 21 22 23
```

In this and later examples, the user's typing is shown in boldface following the S prompt character **>**, and is followed by the response from S. The result of an expression, if not assigned or otherwise used, is normally printed on the terminal.

The operation of giving a name to a data structure is called *assignment*. Assignment is performed by an operator, which will appear as an arrow, " $\leftarrow$ ", in this manual. The arrow may be typed as an underline character or as the two characters " $<-$ ".

(less-than, minus).† On the left of the arrow is a name, on the right any expression. Names can contain letters, digits, and ".", and must start with a letter. Users can choose names for their convenience and can re-use names arbitrarily. The expression is stored on the working database, with the given name; e.g.,

```
> x ← 9:23
```

The sequence of characters "x ←" is normally read as "x gets". Notice that the result of the assigned expression is not printed.

Subsets of data can be extracted by:

```
x[ expression ]
```

The expression defining the subset can have three forms.

- (1) a vector of positive values; the corresponding elements of x will appear in the result; for example, $x[1]$ to extract the first value from x , or $x[1:5]$ for the first 5 values.
- (2) a logical expression of the same length as x ; the elements of x for which the logical expression is true will be selected; for example, $x[x > 0]$.
- (3) a vector of negative values; all elements except those specified will be selected; for example, $x[-1]$ gives all but the first element of x .

Subset expressions can also be assigned to, with the effect of replacing the corresponding elements of the vector:

```
x[ x < 0 ] ← 0
```

replaces all negative elements with 0.

Most of the statistical, graphical and other analyses in S are done by *functions*; e.g.,

```
mean(x)      # arithmetic mean
plot(x,y)    # scatter plot of y versus x
c(1,-2.1,3.2,5.78) # combine data values to make a vector
rnorm(5)     # generate sample of size 5 from standard normal
```

In general, function calls have the form

```
name(arg1, arg2, ... )
```

where *name* is the name of the function, and *arg1*, etc. are any expressions, to be taken as arguments to the function. For example *mean(x)* calls the function *mean* to compute the mean value of the data named x . Typically, functions will have a few basic arguments (supplying the data on which the function operates, for example). These must always be supplied. There may also be any number of optional arguments, which usually give special features or options to control the function in more detail. These may be omitted, with the function taking some default action.

Usually, optional arguments are most conveniently supplied in the form *name=value*, where *name* is the name of the argument (listed in the detailed documentation for the function) and *value* is any expression. The user need only supply the options of interest, in any order. To make it easier to use named arguments, the

† By the way, the expression typed " $x < -3$ " is different from the expression typed " $x < -3$ "; the first is an assignment, the other a comparison.

name can be abbreviated to its first few characters, as long as it can be distinguished from other argument names to the same function. As an example of the use of named arguments, one of many optional arguments to *plot* is *ylab*, a label for the y-axis of the plot:

```
plot(x,log(y),ylab="Logged response values")
```

Note the distinction between the specifying of arguments with "=", testing equality with "==", and the assignment of data with "←". The specification of arguments is local to the single function call; it has no effect on the database.

S also has functions which take an arbitrary number of arguments. For example, the combine function, *c(arg1, ..., argn)* combines all the data in an arbitrary number of arguments to form one vector. These arguments will not generally have names; however, functions with arbitrarily many arguments often recognize special arguments *only* when the name is given.

There are a large number of functions in S, to be introduced gradually in the sections of this guide. All of them can be used with the form of expression illustrated in this section.

2.1.2. Interacting with S: Errors; Interrupts; Help.

As a general approach, users of an interactive system like S should not be afraid to take a chance with a new kind of expression. The language and the individual functions are designed to allow as much flexibility as possible. The effect of errors when they do occur is not usually serious. In interactive analysis, errors tend to be detected quickly, and the correction can be made directly.

Errors may be less likely, particularly when first using S, if complicated expressions are evaluated in a sequence of simpler steps, assigning and reusing the intermediate results. This has the added advantage that the intermediate results are available for examination, to see that they were computed as intended. As you become more accustomed to S, complicated expressions will hold fewer terrors. Also, quite complicated expressions can be useful when turned into source files or macros (section 3.1) so that only a simple macro call is actually typed. Long expressions can be continued over any number of lines, so long as it is clear that the end of the expression can not have occurred yet. This can be assured by ending the line with an operator or a comma in an argument list, or by not supplying enough right parentheses to close the expression; e.g.,

```
> z ← x+3.14159*(y-  
+ mean(y,trim=.1)) # previous line continued
```

The first line was doubly sure of being continued since it ended with the operator "-", and also had an un-matched left parenthesis. As shown above, S prompts with "+" for continuation lines, and comments can be inserted in expressions, from the character "#" to the end of the line.

Errors in S expressions may be detected because what was typed could not be interpreted as an S expression or because one of the functions encountered an error during evaluation. The first type of error (a *syntax* error), will be reported, by typing back the offending line, with the apparent location of the error marked:


```
> x+7,5 # I meant 7.5
```

```
Syntax error
x+7,5
```

The second type of error (an *execution* error), is reported by the function which could not compute the desired result:

```
> 1+"abc"
```

```
Error: attempt to use character data in arithmetic
Error in +
```

In either case, a message will be printed to describe the error, execution of the current expression will terminate, and S will prompt for further expressions. If an error occurs in a long expression, the expression is written to a file, and the system types "Dumped" on the terminal. The expression can be retyped directly, or the user can type

```
> edit
```

which invokes a text editor[†] to allow changes to the expression. The user writes the corrected expression and quits from the editor, after which the expression is automatically re-executed (see *edit* in the detailed documentation).

The user may also interrupt execution of any expression by hitting the key used to issue interrupts or breaks. For example, you may wish to interrupt long printouts or the execution of an expression which was incorrect. If the expression was long enough, the message "Dumped" will be printed and *edit* can be used to correct any mistakes. The expression can be re-executed without editing by typing *again* after the "Dumped" message.

Errors in functions often occur when a mistake is made in the name, number or form of arguments. To provide information on functions, the detailed documentation for all functions is available on-line; the expression

```
help("plot")
```

for example, causes the detailed documentation for the function *plot* to be printed on the user's terminal. Invoking *help* without arguments produces documentation for *help* itself.

Problems that persist after reading the documentation and experimenting may be reported by sending *mail* to a special user at each installation. The S *mail* function prompts the user for text, terminated by a line consisting solely of a period. Mail can also be used to report suggestions or other comments about S. Assuming someone is opening letters at the other end, the user should receive a reply through the operating system's mail command. See the operating system's manual under *mail* for instructions on how to read mail sent to you.

A facility is provided to allow the S user to communicate with the operating system. Any line that begins with the character "!" is passed, untouched, to the operating

[†] Interactive computing inevitably requires some editing of files. If you are not familiar with a text editor, look through "A Tutorial Introduction to the UNIX Text Editor" by B. W. Kernighan, for a discussion of the more commonly needed features.

system for execution. This facility can be used to list files, to enter the text editor, etc. Operating system commands (e.g., text editors) that provide interaction, return to the S environment when that interaction is complete.

```
> !ed mydata
... edit file mydata with text editor ...
q
> # ready for next S expression
```

2.2. Data.

2.2.1. Data Values.

Data values are numbers (real or integer), logical values (TRUE or FALSE), or character strings. Explicit numerical values may be written in the usual integer, decimal fraction or exponent notation:

```
3, -5, 6.272, 3.4e10
```

Internal blanks are *not* allowed in numbers.

Character strings may be specified by any string of legal characters, enclosed in matching quotes (") or apostrophes ('):

```
"Now is the time ..."; '+-*/'
```

The enclosing quotes or apostrophes are only delimiters, not part of the string. A back-slash character may be used to enter a character which would not otherwise be legal, typically a quote or apostrophe:

```
"You are reading the \"S User's Guide\"" # or equivalently
'You are reading the "S User\'s Guide"'
```

Logical data may be entered as T or TRUE or as F or FALSE.

The special data value, NA, stands for Not Available. It may appear wherever a numerical or logical value is expected. The best way to think of this is as a condition holding whenever one wants to indicate missing data. Not all functions in S know how to handle missing values; if data with NAs is given to these functions, an error will result. The function *NA(x)* returns logical TRUE values wherever *x* has missing values, so

```
x[ !NA(x) ]
```

will return the data in *x* with missing values removed. Consider, however, whether this is sensible from the point of the data analysis.

2.2.2. Vectors.

S supports an unlimited variety of data structures. The simplest are *vectors* containing numbers, logical values, or character strings. Some examples are:

```
> 1:3; Weekdays; Month.length > 30
1 2 3
"Monday" "Tuesday" "Wednesday" "Thursday" "Friday"
T F T F T F T T F T F T
```

All S data structures are self-describing; that is, they contain enough built-in

information so that S functions can interpret their length, etc. The user never needs to supply this information in S expressions (except possibly when creating a matrix or other structure from unstructured data values).

Vectors are created either as the result of expressions or when data is read into S (section 2.2.3). Vectors, like all S data structures, are self-describing so that the user need not supply the length, etc. to S functions using the vector. There are, however, functions in S to extract attributes of vectors and other data structures. For example, the function *len(x)* returns the number of data values in its argument. The function *mode(x)* returns the (numerical code for) the mode of the data in *x*. See section 2.2.5 for other attributes of matrices and time-series.

2.2.3. Data Input.

Data values can be read from a file or from the user's terminal to form a vector, by the function *read*:

```
x ← read("myfile")
```

reads values from the file named "myfile". Beginning users should try to keep clear the distinction between a *file* which is generated in the operating system, outside of S, and a *dataset* in S, as generated by an assignment. The file used in the expression above might contain, for example,

```
3.5  2.1
7.893 1.3e10  0  0  0
1
-37.892  80000
```

The vector assigned would be of length 10, with the values shown.

The *read* function assumes that the file consists of data items separated by white space (that is, by blanks, tabs or new lines). If the file argument is not given, *read* will read items from the terminal. In this case the user is prompted with the index of the next item to be read:

```
> x← read()
1: 1  2.5  3.14159
4:
```

3 items read

The empty line terminates terminal input (but not input from a file). Reading directly from the terminal is reasonable for small amounts of data. In addition the functions *append* and *replace* provide a facility for data editing within S. However, for larger datasets, most users will find it preferable to have the data items on a file, so that a text editor can be used to edit out typing errors before input.

If the user wants to control the number of items read from the file, or if more than the default maximum of 2000 data items is to be read, the optional argument *len* should be given:

```
read("myfile",len=7)
```

reads only the first two lines (seven items) of the file shown.

The input data may also be character strings. By default, *read* examines the first item; if it is numeric, all following fields are assumed to be numeric. Otherwise, the

items are all assumed to be character strings. In addition one can force the input to be treated as character strings by using the optional argument, *mode*:

```
read("file2",mode=CHAR)
```

Because *read* looks for white space to separate items, character strings which contain blanks or tabs must be enclosed in quotes:

```
Saturday night and Sunday morning
```

reads as five character string items, while

```
"Saturday night and Sunday morning"
```

reads as one.

An item appearing as NA on input will generate a corresponding missing value in a numeric vector. NAs are not allowed in character vectors, although a character string can be empty, i.e., "".

As illustrated in this section, the basic data for input to S is a file of human-readable information, divided into fields by blanks or other separators. Techniques for dealing with different types of files and other forms of data transfer will be discussed in section 3.2.4.

2.2.4. Databases.

Data structures assigned names in S expressions become part of a *database*; that is, a collection of datasets that are retrieved automatically when referred to by name in an S expression. Three databases are available automatically in each S session. The first is the user's *working* database. Datasets created by assignments are stored here. The second database is the user's *saved* data. This is intended for data of permanent value. Placing datasets here is a useful safeguard against accidentally assigning some other dataset the same name. The expression

```
save(arg1, ...)
```

puts its arguments onto the save database. If the argument is of the form *name=value* then *name* is used as the name of the saved dataset. Otherwise, the argument itself should be a dataset (presumably not currently on the save database) and the saved data will have the same name.

The working data and save data are local to the current user. A third database is the S *shared* database, containing datasets of frequent value to users in general, including some classic statistical examples and standard values (e.g., the names of the states, months, etc.)

The available databases form a list maintained for each user. When a dataset name appears in an expression, the databases are searched in order for the corresponding name until the dataset is found or the search list is exhausted. Notice, in particular, that a dataset will not be retrieved from the save database or the shared database if a set of the same name appears on the working data.

When data has been moved to the save database, or is for some other reason not needed, it may be removed from the working database by

```
rm(x)
```

To obtain a list of the names of all datasets in the working data use the function *list*.

Functions such as *list*, *save* and *rm* can be made to refer to any of the databases currently available by use of the optional argument *pos*. This specifies the position of the database in the search list; by default, positions 1, 2, and 3 correspond to working, save and shared databases. For example:

```
> rm(x,pos=2) # remove x from save database
> list(pos=2) # what's on the save database?
  "city"      "label"      "median"    "quartile"  "xy"
> list # working database
  "i"  "tx"  "ty"  "tz"  "xl"  "yl"
```

2.2.5. Data Structures: Matrices and Time-Series.

General data structures in S are built up from vectors, and are usually the result of functions that return more than a simple vector. Some data structures are so widely used that they are essentially built into the language. Two examples are *matrices* and *time-series*.

Matrices contain, in addition to a vector of data values, information defining the number of rows and number of columns. Although matrices are actually a special case of general multiway arrays, they are so common that we will introduce them here, and defer to section 3.2 the discussion of general arrays. Subsets of matrices may be specified by two subscript expressions, $x[i1, i2]$, where *i1* and *i2* define subsets of rows and of columns. If either expression is omitted, all the corresponding rows or columns are included. For example,

```
x[, c(1,3)] # the first & third cols
x[res>.5,]  # all rows i where res[i]>.5
x[1:3,2]    # x[1,2], x[2,2], x[3,2]
```

Matrices are produced as the result of many functions in S. They may also be generated from a vector of data, using the function *matrix*. Besides the data to be used, *matrix* takes the number of rows and columns desired in the matrix. For example

```
> matrix(1:12, 3, 4)
```

Array:
3 by 4

	[, 1]	[, 2]	[, 3]	[, 4]
[1 ,]	1	4	7	10
[2 ,]	2	5	8	11
[3 ,]	3	6	9	12

By default, *matrix* assumes the data values are given one column at a time. When reading data from a file with the values laid out in rows, the optional logical argument *byrow* is used:

```
> x ← matrix( read("matfile"),ncol=5,byrow=TRUE )
```

Notice that only one of the number of rows or number of columns need be specified. The other is inferred from the number of data values.

A number of S functions create, manipulate and give attributes for matrices. The number of rows and the number of columns are attributes obtainable from the functions *nrow(x)* and *ncol(x)*. It is possible, of course, to create matrices by extracting

rows and columns from an existing matrix. In the other direction, vectors and matrices may be bound together as either the rows or columns of a new matrix. For example, suppose *xold* is a matrix and *update* is a vector:

```
xnew ← cbind(1, xold, update)
```

creates a new matrix with a first column of all ones, the next columns the same as the columns of *xold* and the last column taken as *update*. In general, *cbind* can have any number of arguments, any of which may be vectors or matrices. All matrices should have the same number of rows. Notice that the single value, 1, was expanded to a vector of length equal to the number of rows of *x*. The result will have the same number of rows as the matrix arguments (or the length of the longest vector, if there were no matrix arguments). The function *rbind* works similarly with rows:

```
xnew ← rbind( xold, newcols )
```

binds the data in the arguments to *rbind* together as rows of the result.

Functions which create new matrices or vectors from existing matrices are *t* (the transpose of *x*), *diag(x)* (the vector of the diagonal elements of *x*), *row(x)* and *col(x)* (matrices of the same shape as *x*, but filled with the row number and the column number). For example, suppose *x* ← *matrix(1:12,3,4)*, as in the example above.

```
> t(x)
  1 2 3
  4 5 6
  7 8 9
10 11 12
```

```
> diag(x)
  1 5 9
```

```
> row(x)
  1 1 1 1
  2 2 2 2
  3 3 3 3
```

The functions *row* and *col* are useful for selecting special subsets of a matrix:

```
x[ row(x) >= col(x) ] ← 0
```

zeroes the lower triangle of *x*. The function *diag* also creates matrices given the diagonal elements. If *values* is a vector, *diag(values)* is a square matrix, the number of rows and columns equal to *len(values)*, with the diagonal filled in from *values*, and all other elements 0. If only 1 value is given to *diag*, it creates an identity matrix of this order; e.g., *diag(10)* is a 10 by 10 matrix with 1 on the diagonal and 0 elsewhere.

Given vectors of character strings which label the rows and/or columns of a matrix, a specialized and more informative printout can be generated from the *print* function.

```
print(votes.repub, rowlab=state.abb, collab=encode(votes.year))
```

The special arguments *rowlab* and *collab* to *print* are used in place of numerical labels to label the rows and columns, respectively. Note that the labels may not be numeric: see section 3.3.2 and the detailed documentation for the *encode* function for general ways to construct informative labels.

Time-series are defined by a vector of data values, considered to be taken at equally spaced times. Three time-series parameters are used to define the corresponding time domain implicitly: *start*, *end* and *nper* for the times at which the observations start and end and the number of observations per unit time period. Conventionally, time-series data are used most often relative to units of one year (typically annual, quarterly or monthly). For example, a monthly series starting in June of 1960 could be specified from data on a file "co2data" through the function *ts* as follows:

```
> co2 ← ts( read("co2data"), start=c(1960,6), nper=12 )
```

Notice that, as with *matrix*, we gave only enough time parameters so that the remainder could be computed from the number of data values. Also note that the vector *c(1960,6)* represents June, 1960.

Monthly and quarterly series are printed with special formats.

```
> co2
```

Time-series:

	Jan	Feb	Mar	Apr	May	Jun
1960						319.74
1961	316.97	317.74	318.63	319.43	320.47	319.71
1962	318.06	318.59	319.74	320.63	321.21	320.83
1963	318.80	319.08	320.15	321.49	322.25	321.50
	Jul	Aug	Sep	Oct	Nov	Dec
1960	318.15	316.00	314.23	314.07	315.04	316.19
1961	318.78	316.84	315.16	315.56	316.14	317.13
1962	319.55	317.75	316.27	315.62	316.84	317.70
1963	319.67	317.61	316.25	316.17	317.01	318.36

Many functions in S know how to operate on time-series; in particular, arithmetic operators operate on two time-series to produce a result defined on the intersection of the two time domains.

Special S functions create, manipulate and obtain attributes of time-series. The functions *start(x)*, *end(x)*, *nper(x)* give the starting time, the end time and the number of observations per unit time of *x*. The function *time(x)* returns a time series on the same domain as *x*, with values equal to the corresponding time. Similarly, the function *cycle(x)* returns a time series whose values are the period (within the unit of time) corresponding to each observation. For example suppose *x* is a monthly series starting in February 1979 and ending in May 1980. The values in *cycle(x)* are

```
2, 3, 4, ..., 12, 1, 2, 3, 4, 5.
```

For some applications (regression, for example) it is helpful to bind together time-series as columns of a matrix. The function *cbind* is not suitable if the series have different time domains, since it ignores the time-series nature of the arguments. The function

```
tsmatrix(x1, x2, ...)
```

binds its arguments as columns, but computes the intersection of all the time domains if any of the arguments is a time-series.

Matrices, time-series and other data structures with vectors of data as components are treated as ordinary vectors by certain functions. For example, the function *sort* returns a vector of sorted data values. Even if the argument is a matrix or time-series, the result is no longer a similar structure. Similarly, subsets of matrices and time-series may be selected just as if they were vectors, in which case the result is a vector. For example,

```
co2[ co2>mean(co2) ]
```

produces the vector of all values in *co2* which are larger than the mean value. The function

```
window(x, start, end)
```

can be used to produce time-series subsets of time-series.

Functions written to accept matrices or time-series can be given vectors as arguments. Vectors will be interpreted as matrices with one column, or as time-series starting at time 1 with one observation per period.

General data structures arise as the result of statistical and other functions. When the result of a function consists of several vectors, the function will return a structure whose *components* are the individual vectors (or, equally, other data structures) making up the result. For example the function *regress* and a number of other functions fit linear models to data. In this case, a number of components are returned, including the components named *coef* for the vector of coefficients and *resid* for the residuals from the fit. Structure in the input data is preserved in the residuals: a regression to fit a model to a time-series produces residuals which are also a time-series.

Components of a structure may be extracted by the operator "\$". For example, if one assigns the result of a regression, by $z \leftarrow \text{regress}(x,y)$, then the vector of coefficients is obtained as:

```
z$coef
```

The expression on the left of "\$" is usually a dataset name, but it may also be a function call or parenthesized expression (if the rest of the structure is not needed), for example:

```
regress(x,y)$coef
```

does the regression, extracts the coefficients, and throws the rest of the regression structure away. The detailed documentation in section 9 describes the components of the data structures returned by S functions. As in the case of named arguments to functions, it is only necessary to give enough characters of a component name to identify the component uniquely.

2.3. Data Analysis.

2.3.1. Statistical Functions.

A few of the more common statistical functions are described here. Elementary estimates of location are:

```
mean(x); median(x)
```

for arithmetic mean and median. The mean can be extended to trim a fraction of the

largest and smallest values by the optional argument *trim*. Sample variances, covariances and correlations are given by:

```
var(x); var(x,y); cor(x,y)
```

If the arguments are matrices, the results are covariance and correlation matrices, treating the columns of *x* and *y* as variables.

A model-fitting function is

```
regress(x,y)
```

which does a linear least-squares regression and, by default, prints out a summary including significance-test statistics and the covariance matrix of the coefficients. The data structure returned contains residuals, coefficients and numerical summaries. Optional arguments to *regress* allow fitting without an intercept term or fitting by weighted least-squares.

Related functions *lfit* and *rbiwt* also fit linear models, but not by least squares. The former fits a general linear model by minimizing the sum of absolute residuals. The latter fits a simple linear model (only one independent variable) by a robust criterion, essentially trying to down-weight the effect on the fit of observations with large residuals. In some circumstances, either of these methods is preferable to least squares, if there may be a few deviant observations or if the assumption of normal distribution of errors is not appropriate. All the linear fitting functions return components *coef* and *resid* as discussed above.

The function *twoway* fits an additive model to a two-way array; i.e.,

$$x[i,j] = \text{grand} + \text{row}[i] + \text{col}[j] + \text{resid}[i,j]$$

The fit is usually iterative, using medians or trimmed means. The structure returned by *twoway* has components *grand*, *row*, *col*, and *resid* as suggested above. Optional arguments provide control over the method and the number of iterations.

It is sometimes useful to generate pseudorandom values from a specified statistical distribution; e.g., as tests for some analysis or to compare with actual observations. S contains random number generators for a number of distributions. They all have names consisting of the letter "r" followed by a code for the distribution, such as "norm" for the normal, "unif" for the uniform and "t" for Students t-distribution. The first argument is the sample size desired. Other arguments, if any, are the parameters of the distribution. Thus,

```
> xn ← rnorm(100); xt ← rt(75,2)
```

stores in *xn* a sample of 100 from the standard normal and in *xt* a sample of 75 from the t-distribution with parameter (degrees of freedom) equal to 2. The codes for the distributions are:

Code	Distribution
beta	beta
cauchy	Cauchy
chisq	chi-square
f	f
gamma	gamma
lnorm	log-normal
logis	logistic
norm	normal
stab	stable
t	Student's t
unif	uniform

For information about the parameters to the distributions (including default values), see the detailed documentation under the corresponding code. See also the dataset documentation for "Random.seed" in section 9d for a useful technique for reproducing previous random samples.

In addition to random number generators for these distributions, there are also probability and quantile functions named by placing the letter "p" or "q" before the distribution code. (See section 3.3.1.2).

2.3.2. Data Manipulation.

The function *c* combines all its arguments (treated as vectors) into a single vector. It is the usual way of making small vectors inside expressions.

```
> c(1:3, 5, 7, 9)
1 2 3 5 7 9
```

The function *rep* replicates its first argument as many times as specified by its second argument:

```
> rep(1:3,2)
1 2 3 1 2 3
```

Use of *rep* is a good way to fill out data structures with patterned values; e.g.,

```
rep(c("Low", "Mid", "High"), c(3, 6, 5))
```

makes a character vector with 3 repetitions of "Low", 6 of "Mid", and 5 of "High".

The function *seq* is a general form of the *:* operator introduced earlier. The expression *m:n* is equivalent to *seq(m,n)*, but there are also one-argument and three-argument forms. The expression *seq(x)* returns the vector *1:len(x)* or the vector *1:x* if *x* is a single value. This is convenient for manipulating subsets of vectors; for example, an expression to determine which elements of *x* are negative would be:

```
seq(x)[ x < 0 ]
```

With three arguments, *seq* interprets the third argument as an amount to step in producing the sequence going from the first argument to the second.

```
> seq(-1, 1, .1)
-1., -.9, -.8, ..., .8, .9, 1.
```


The function *sort* returns the vector of sorted numeric or character-string data values corresponding to its argument: *sort(x)* returns the values in *x* in ascending or alphabetical order.

```
> x ← c(1,-1,2,-2,.5); sort(x); sort(Weekdays)
-2.0 -1.0 0.5 1.0 2.0
"Friday" "Monday" "Thursday" "Tuesday" "Wednesday"
```

Two other functions are closely related to *sort*: *order*, which returns the ordering permutation, i.e.,

```
x [ order(x) ] # is the same as sort( x )
```

and *rank*, which returns the positions of the original elements in the sorted data and takes account of ties. The three functions are very powerful in doing data analysis which depends on sorting or on finding large or small values. It is also easy to confuse the roles of these functions; some examples of their common use may help. The *order* function is used most frequently to permute other data consistently with the result of sorting one set of data.

```
> xorder ← order( x ); xorder
4 2 5 1 3
> a ← x[xorder] #sorted version of x
> b ← y[xorder] #y in the same order
```

Another capability of *order* is sorting on several fields. For example,

```
order(last.name, first.name)
```

gives an ordering that is alphabetical by name. If two last names are identical, the first names determine the ordering.

The *rank* function may be used to generate a number of statistical summaries based only on the ordering of a set of data, not on the numerical values. These are commonly called *distribution-free* or *nonparametric* statistics. For example, the *two-sample rank test* uses a test statistic consisting of the sum of the ranks for one set of data, say *x*, when both *x* and *y* are ranked together. The expression *rank(c(x,y))* gives a vector containing the ranks taken together; the first *len(x)* elements of this are the ranks associated with *x*. A single expression to give the desired statistic is then:

```
sum( rank(c(x,y))[1:len(x)] )
```

Notice that the sorting routines sort in ascending order. To get sorts or orderings in *descending* order, use the reversing function *rev*; for example, *rev(sort(x))* sorts *x* in descending order.

As illustrated by the example using *order*, the subset operation can re-arrange a vector. It can also be used to replicate data values in a vector, or the rows and columns of a matrix:

```
x[ c(1,1,3:9,1,1) ]
```

would contain four copies of the first item of *x*, along with the third through the ninth items.

The function *match(x,y)* tries to find each element of *x* in *y*. The *i*th value in the result is the position of *x[i]* in *y* (or NA if *x[i]* was not found in *y*). There are two useful applications for *match*. In one the second argument is a set of possible values

and the purpose is to turn the values in x into indices in this set. Suppose, for example, that the data in x was coded "L", "M" or "H" for low, medium or high. For analysis, we prefer numerical values. Then,

```
match(x,c("L","M","H"))
```

is a vector that codes the characters as 1, 2 and 3. This is very similar to the operation carried out by the function *code*, described in section 3.2.6.

The second use is closely related to the *rank* and *order* functions above. In this x and y represent versions of the same data (or nearly so), but in a different order. If we do not know the relation between the two sets, then *match* can be used to find it. Specifically, if x and y have exactly the same values, *match*(x,y) is the permutation which orders y the same way as x . For example suppose *names* is a vector of labels for some observations, in the desired order. Now some new data, say a matrix *update*, is obtained, but the observations are in a different order, specified by *newnames*. Then the new data can be put into the old order by

```
update[ match(names,newnames), ]
```

Notice that *match* matches the *old* names to the new. Also, this expression has a built-in check that all the observations in *names* are included in *newnames*; otherwise, the subset expression will contain NAs and an error will be generated. Of related interest is the function *unique*(x), which returns all the unique values in x . In applications like the previous example, one would want to keep *names* as a unique list of names (e.g., so that a correct count of the number of names would be given by *len*(*names*)).

The functions *all* and *any* take logical vectors and return a single TRUE or FALSE according to whether any or all of the values are TRUE. For example,

```
all( x >= 0 ); any( x < 0 )
```

return TRUE and FALSE, respectively, if there are no negative values in x .

A number of special data-manipulation functions are available for time-series. If x is a time-series, the following data manipulation functions are available.

```
smooth(x)    #smooth the data values
diff(x)      #forward differences
lag(x,k)     #same data occurring k periods earlier
```

The smoother is a robust smoother, made up of repeated simple smoothing operations (see the detailed documentation). The series *diff*(x) has the same starting date, but has length *len*(x)-1. The lagged series, in contrast, has the same number of data values with the starting and ending times altered.

2.3.3. Numerical Calculations.

The S language includes the standard mathematical functions:

```
sin  cos  exp  log  log10
asin acos  atan sqrt
```

These functions transform a single vector or structure by the corresponding mathematical function. Operations on matrices and time-series yield matrices and time-series as values.

There are also a number of functions which compute elementary numerical results:

ceiling floor trunc round

ceiling and *floor* find the closest integer values not less than and not greater than the corresponding values in their arguments. The function *trunc* truncates values towards zero, while *round* rounds values to the nearest integral value. Optionally, *round* can take a second argument, giving the number of digits after the decimal to which rounding takes place. Some examples:

```
> x
-1.90691 0.76018 -0.26556 -1.89828 0.08571
> ceiling(x); floor(x); trunc(x); round(x); round(x,1)
-1      1      0     -1      1
-2      0     -1     -2      0
-1      0      0     -1      0
-2      1      0     -2      0
-1.9    0.8   -0.3   -1.9    0.1
```

There are also functions to add and multiply together all the elements of a vector: *sum(x)* and *prod(x)*, which return the single number giving the sum or product. The functions *max* and *min* give the largest and smallest values in their arguments (these two functions, however, can have an arbitrary number of arguments). See also the related functions *pmax* and *pmin*. The function *range* returns a vector with two numbers which are the max and min of all of its arguments. That is, *range(x,y)* is equivalent to *c(min(x,y),max(x,y))*.

The function *diff* mentioned in section 2.3.2 can also be used on arbitrary numerical data to produce the first differences. The function *cumsum* is the inverse of *diff*; that is, the *i*-th element of *cumsum(x)* is the sum of the first *i* elements of *x*. For example, if *x* (*<- 1:10*), then *diff(x)* is a vector of 10 ones and *x == cumsum(rep(1,10))*.

2.4. Graphics.

The combination of interactive computing with graphics provides a powerful environment for data analysis. In S there are a number of high-level plotting functions, along with other facilities for adding to plots or controlling their appearance in detail.

Before using any of the graphics functions, the user must specify the type of graphics terminal or device which will be used for plotting. The interactive terminals supported by S, and the S commands which specify the device are as follows:

Terminal Type

S Command

Hewlett-Packard 7221

hp72f, hp72h, or hp72v

Printer

printer

Tektronix (4006, 4010)

tek10

Tektronix 4012

tek12

Tektronix 4014

tek14, tek14q

Tektronix 4662

tek46f, tek46h, tek46v

The *printer* device will work on any terminal, but gives considerably less attractive

plots than the true graphics terminals. Three device commands are available for the pen-plotting terminals (Tektronix 4662 or Hewlett-Packard 7221) corresponding to large paper, or to letter-size paper oriented horizontally or vertically. A device command must be given once before doing any plotting. The device may be changed during the terminal session (for example, if changing paper size on the pen plotters) by giving another device command later.

The most frequently useful high-level plotting routine is

```
plot(x,y)
```

which produces a scatter plot of the two sets of data[†]. This function can also be used to produce a time-series plot, if only one dataset is given. If a single argument is given, the x-axis will show time, and the y-axis the data values. (If the argument is not a time-series, the time will be `1:len(x)`.) The function `plot` has an optional argument, `type`, which can be used to produce a plot with lines connecting the points, rather than characters plotted at the points:

```
plot(y,type="l")
```

gives a time-series plot of `y` with lines joining the successive values. (The character inside the quotes is the letter *ell* standing for "lines", and not the digit one.) Notice that the second argument is given in the `name=expression` form, so that `plot` will realize that only one set of data was given. Other values for `type` include "b" (both) to plot both points and lines. The default value is "p" (points), for a scatter of points.

The example plot shown here was the result of the call:

```
plot(fitted,abs(z$resid),ylab="Absolute residuals")
```

Logarithmic transformations (to base 10) of either or both axes may be done automatically:

```
plot(fitted,abs(z$resid),log="y")
```

will use a log scale for `y`. This differs from plotting `log10(y)` since logarithmic axes are generated. Similarly, values of "x" give a log scale for x-axis, and "xy" or "yx" give a log scale for both.

While `plot` will plot a single time-series, as above, there is a special high-level routine which plots any number of time-series:

```
tsplot(y1,y2, ...)
```

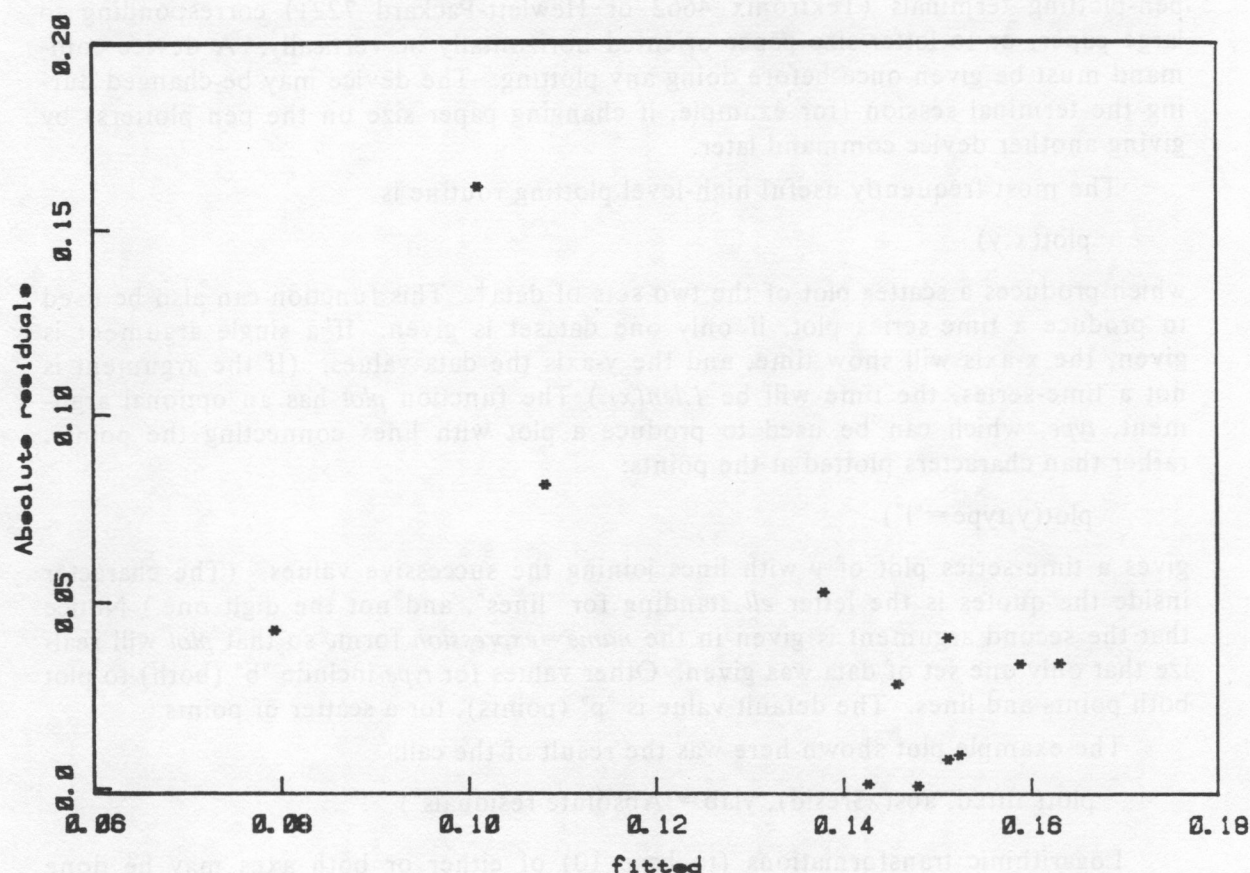
The x-axis will be a range of times which includes the range of all the arguments. For this plot, the default type is "l", connected lines. Other types may be given as above. When there are several series given, the function uses changes in line texture (dashed lines), in color, and in the plotted character to differentiate the series.

A similar function plots one or more columns of a matrix `y`, against corresponding columns of another matrix, `x`:

```
matplot(x,y)
```

The type is "p" by default, and different plotting characters, line textures and colors are again used. If the arguments differ in number of columns, the function will reuse

[†] All S functions doing plotting follow the convention of giving the arguments `x,y` to specify data to plot, even though one might say "Plot `y` vs `x`." The consistency (which is shared with model-fitting functions like `regress`) is convenient and eventually easier to remember.



columns cyclically. In particular, either of the arguments may be a vector.

A different high-level plotting function plots a histogram of a single set of data:
`hist(x)`

For optional arguments, see the detailed documentation. There is an analogous routine for non-graphics terminals:

`stem(x)`

which prints a stem-and-leaf display. This is essentially a histogram with bars shown horizontally. In addition, the bars are made up of digits to give (to limited accuracy) the individual values in the data. Other summary information is also printed: see the detailed documentation.

A powerful graphical tool for looking at probability distributions is the *probability* or *quantile-quantile (q-q) plot*. The S function

`qqplot(x, y)`

produces an *empirical q-q plot* useful for investigating whether x and y appear to be samples from the same underlying distribution. The function

`qqnorm(x)`

produces a *normal q-q plot* that plots the sorted data x against quantiles of the standard normal distribution. Plots with relatively large or systematic deviations from a straight line give evidence of non-normality.

Once you have a plot, it can be titled by

```
title(main,sub,xlab,ylab)
```

which places a large title on top, a sub-title at the bottom, and labels on the axes. For example:

```
title("Iris Data Analysis","Figure 3",
      "Sepal Length","Petal Length")
```

Many of the high-level functions will automatically provide x and y labels based on the names of the plotted datasets. In this case, simply omit *xlab* and *ylab* from *title*.

Points, lines and text may be added to an existing plot through the functions *points*, *lines* and *text*. For example, to plot a time-series and add a smoothed curve to the plot:

```
> plot(co2); lines(smooth(co2),col=2,lty=2)
```

We specified values for the graphical parameters *col* and *lty* to alter the color and line-type for the plotted curve. In using the *points* function, parameter *pch* may be useful in specifying an alternative plotting character; for example, *points(x,y,pch="+")*. For plots with a large number of points, either through *points* or one of the high-level functions, one may prefer a plotting dot to a character. For a centered plotting dot, use *pch="."*. For a further description of graphical parameters, see section 3.4. Also, the argument *marks=* to the *points* function uses a set of plotting symbols rather than characters; see the detailed documentation for the set of symbols available.

When using the printer device, each frame is assembled in an internal buffer. To get the buffer to be printed, the user must signal the end of this frame. This may be done either by calling another high-level plotting function (but remember that you will always be one frame behind), or explicitly by the function

```
show
```

2.5. Example.

This "basic" section of the user's guide ends with an example that makes use of many of the ideas presented above. The example should clarify the use of a number of functions in the context of a realistic problem in data analysis. The techniques described in this section should enable you to begin working with S. After reading this far and studying the example, it would be a good idea for you to actually try to use some of the functions presented thus far.

we want to read in data from the file "mydata" that contains data on population, income, illiteracy, and life expectancy for ten states during 1977. the file itself looks like this ...

```
> !cat mydata
3615 3624 2.1 69.05
365 6315 1.5 69.31
2212 4530 1.8 70.55
2110 3378 1.9 70.66
21198 5114 1.1 71.71
2541 4884 0.7 72.06
3100 5348 1.1 72.48
```



```
579 4809 0.9 70.06
8277 4815 1.3 70.66
4931 4091 2.0 68.54
```

the rows correspond to different states, the columns are population, etc.

```
> m ← matrix( read("mydata"), ncol=4,
+   byrow=T ) # read data, create matrix, assign name m
Read 40 items
> m # print the matrix
```

Array:

10 by 4

	[, 1]	[, 2]	[, 3]	[, 4]
[1 ,]	3615	3624	2.1	69.05
[2 ,]	365	6315	1.5	69.31
[3 ,]	2212	4530	1.8	70.55
[4 ,]	2110	3378	1.9	70.66
[5 ,]	21198	5114	1.1	71.71
[6 ,]	2541	4884	0.7	72.06
[7 ,]	3100	5348	1.1	72.48
[8 ,]	579	4809	0.9	70.06
[9 ,]	8277	4815	1.3	70.66
[10 ,]	4931	4091	2.0	68.54

construct character vector to label the columns

```
> varname ← c("Population (1000s)", "Income", "Illiteracy",
+   "Life Expectancy")
> varname # print it
```

```
      "Population (1000s)"      "Income"      "Illiteracy"
[4] "Life Expectancy"
```

```
> income ← m[,2] # second column is income
> stem(income) # stem and leaf display
```

```
N = 10 Median = 4812.000 Hinges = 4091.000 5114.000
```

Decimal point is 3 place(s) to the right of the colon

```
 2      2      3 : 46
      5      4 : 15889
 3      2      5 : 13
 1      1      6 : 3
```

a state's total income is average per capita income

times its population

```
> sum(m[,1]*m[,2]) # total income from 10 states (thousands of dollars)
232760192
```

```
> list # see what datasets are on working file
"income" "m" "varname"
```

with the text editor, we have already created a file containing the names of our states, called "statenames"

```
> !cat statenames
```

Alabama

Alaska

Arizona

Arkansas

California

Colorado

Connecticut

Delaware

Florida

Georgia

```
> state ← read( "statenames" )
```

Read 10 items

now a pretty print of our data matrix

```
> print(m,rowlab=state,colab=varname)
```

Array:

10 by 4

	Population (1000s)	Income	Illiteracy	Life Expectancy
Alabama	3615	3624	2.1	69.05
Alaska	365	6315	1.5	69.31
Arizona	2212	4530	1.8	70.55
Arkansas	2110	3378	1.9	70.66
California	21198	5114	1.1	71.71
Colorado	2541	4884	0.7	72.06
Connecticut	3100	5348	1.1	72.48
Delaware	579	4809	0.9	70.06
Florida	8277	4815	1.3	70.66
Georgia	4931	4091	2.0	68.54

what states have incomes above the median?

```
> state[ income > median(income) ]
"Alaska" "California" "Colorado" "Connecticut" "Florida"
```

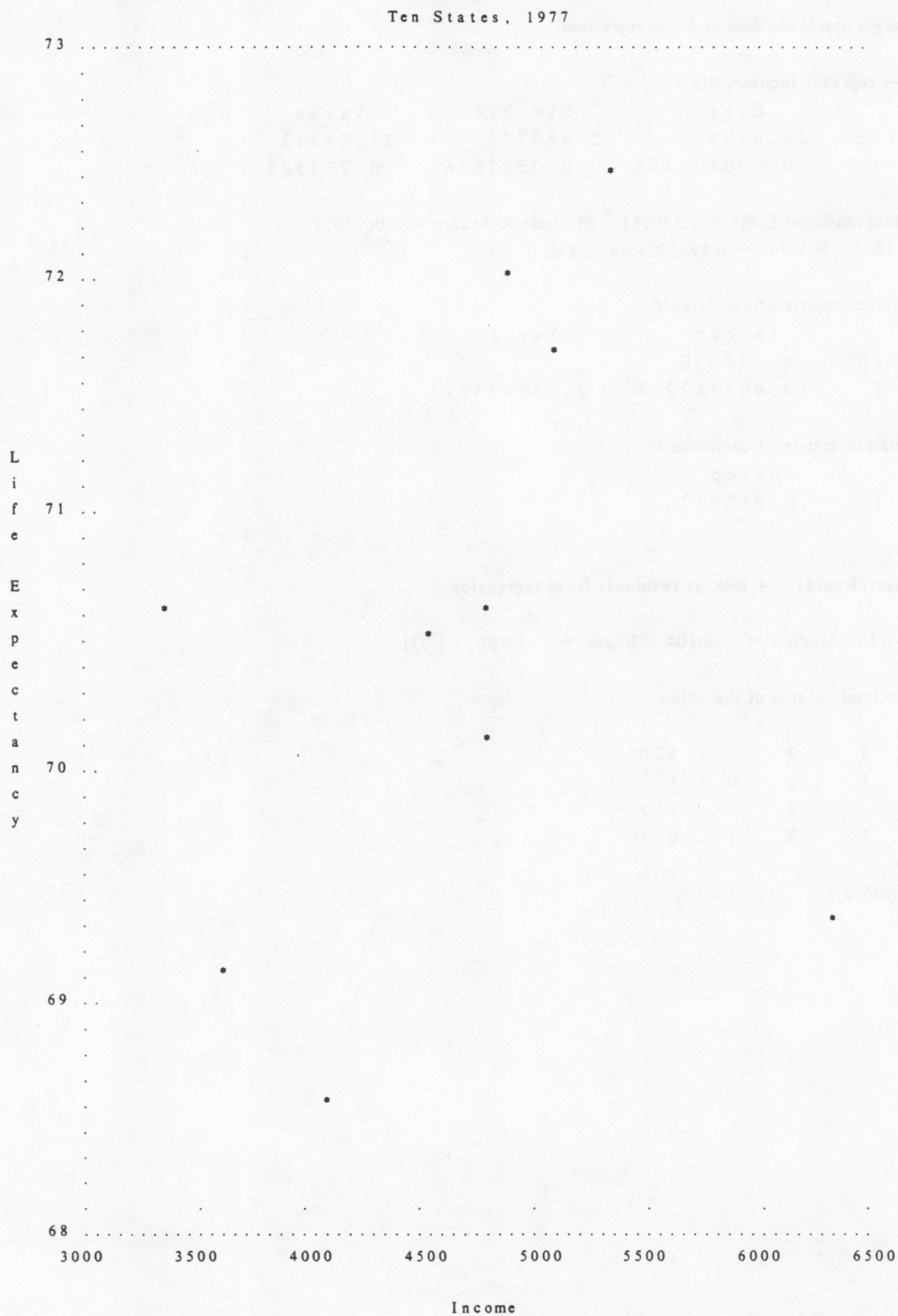

Now some graphics -- give the name of the graphics device

```
> printer # any printing terminal

> life ← m[,4] # life expectancy is 4th column

> plot( income, life, xlab=varname[2], ylab=varname[4],
+      main="Ten States, 1977" )

> show # show the current plot
```



fit straight line to the data by linear regression

> r ← regress(income, life)

	Coef	Std Err	t Value
Intrcp	68.61095	2.488778	27.56812
Var 1	0.000404419	0.0005228263	0.773525

Residual Standard Error = 1.339383 Multiple R-Square = 0.0695879

N = 10 F Value = 0.5983404 on 1, 8 df

Covariance matrix of coefficients

	Intrcp	Var 1
Intrcp	6.194020	
Var 1	-0.001282218	2.733474e-7

Correlation matrix of coefficients

	Intrcp
Var 1	-0.985412

> stem(r\$resid) # look at residuals from regression

N = 10 Median = 0.104 Hinges = -1.027 1.031

Decimal point is at the colon

3	3	-1	:	970
4	1	-0	:	5
	3	0	:	117
2	3	1	:	057

now finished

> q

Advanced Use of S

3. Advanced Use of S.

3.1. Language.

3.1.1. Extending the Language: Source and Sink; Macros.

There are times when an analysis performed with S must be carried out repeatedly; for example, each time some observations are added to a dataset. One may then wish to save the effort of typing the expressions each time. The simplest approach is to use a text editor to put the expressions onto a file, and then use the function *source* to cause S to read its command input from the file:

```
> source("cmdfile")
```

causes expressions to be read from the file "cmdfile". When the whole of the file has been read, S resumes reading commands from the user's terminal. An error or user interrupt will also terminate reading from the source file. Source files may themselves contain source commands; on terminating reading from the second file, S resumes reading the first.

Until you are sure the expressions in a source file are correct, it is good to have S type the expressions on the terminal before evaluating them. The argument *echo* to the function *options* controls this.

```
> options( echo=1 ) # echo expressions from source file
```

We have already shown the *source* feature in a disguised form. The automatic dump and use of *edit* described in section 2.1.2 actually prepares a file which is then executed by a hidden source function.

A similar function, *sink*, directs standard S printing to be sent to a file rather than to the terminal:

```
> sink("outfile")
```

This is mostly useful for editing printed output, preparing examples or archiving results. After *sink()* is typed, results are again printed on the terminal.

The simple use of *source* is limited in that the file contents are completely fixed. Any changes in the expressions to be executed must be edited into the file before execution. Usually, we would like to generate similar, but not identical, expressions; for example, if we want to analyze different datasets in the same way. The need is for something which acts like a new S function, taking arguments to change the expression evaluated. There are two different ways to achieve this in S: by defining a *macro* which expands into a source file, with argument text substituted into the macro definition; and by creating a *new S function*, which is programmed, compiled and loaded into a form exactly like built-in S functions. The second is more efficient in execution and more general, in that it is not restricted to combinations of existing S functions. Macros are usually simpler and quicker to set up or modify. The rest of this subsection introduces briefly the use and definition of macros. Further information on writing macros is in section 4, and writing new functions is described in section 6.

Macros are invoked in S by giving the macro name, preceded by the character "?" and followed by the arguments to the macro, if any, in parentheses. As with S

functions, the arguments are separated by commas and may be given either positionally or by name. The macro processor looks up the definition of the macro as an S data structure, and substitutes the arguments into the macro text. The result is text which is passed to S to be parsed and evaluated, as a hidden source file.

Macros are defined by use of the S function *define*. This takes an optional file name as an argument and reads one or more macro definitions from the file, or from the user's terminal if no file is given. The form of a macro definition is shown by the following example:

```
MACRO uscale(x)
(x-min(x))/(max(x)-min(x))
END
```

This macro returns its argument rescaled to the range 0 to 1. The MACRO statement gives the name of the macro, *uscale*, and specifies the arguments, here just *x*. The END statement terminates the definition. In between comes the text, consisting of any S expressions, possibly including other macro calls. When the macro is called, occurrences of arguments in the macro text are replaced by the corresponding actual arguments. Arguments may be omitted. If default text is to be substituted for a missing argument, this text is specified between slashes in the MACRO statement. For example, the macro definition

```
MACRO uscale( x/mydata/)
```

would cause the macro to substitute *mydata* for the argument *x* in the macro text if the user typed *?uscale* without an argument.

As a result of *define*, a dataset is stored on the save database for each macro defined. The macros are all stored under the prefix "mac.": the dataset corresponding to macro *uscale* is *mac.uscale*. To print a macro definition, call the function *mprint* with the macro dataset name as an argument:

```
> mprint(mac.uscale)
```

```
MACRO uscale(x)
($1-min($1))/(max($1)-min($1))
END
```

Occurrences of the argument names in the text have been replaced, during definition, by \$1, \$2, etc. for the first, second, etc. argument. To edit the macro, call the function *medit*, also with the dataset name as an argument. After completing the edit, write the edited version (without giving a new file name), quit from the editor, and the macro will be redefined.

Suppose the user invokes the macro *uscale* by typing

```
> ?uscale(sqrt(abs(mydata)+1))
```

Then, the expanded macro text would contain four instances of the expression *sqrt(abs(mydata)+1)*. The use of intermediate datasets to contain the values of macro arguments can save repeated evaluation.

```
$Tx ← x
```

as the first expression of the macro definition copies the argument into the dataset *Tx*, ensuring that the argument is evaluated only once. Macros will generally involve several expressions; by making the text into a compound expression (section 3.1.3),

the value of the macro will become the last subexpression. The macro *uscale* could thus be written:

```
MACRO uscale(x)
  ({ $Tx_x; $Tr_range($Tx)
    ($Tx-$Tr[1])/diff($Tr)
  })
END
```

The use of "{" at the beginning and "}" at the end of the text ensures that *uscale* can be used wherever an S function could be used (see section 4.2).

We suggest the notation *\$Tx*, etc. as a way of putting datasets for intermediate results in a fixed place, to avoid clutter or accidental overwriting of useful data, yet still preserving the mnemonic value of the name.[†] (The "\$" character overrides any *prefix* (see section 3.2) and the "T" is a convention to denote a dataset for temporary use).

It is important to understand the distinction between *macro time*, when the expressions involving macro calls are being expanded, and *evaluation time*, when the expanded S expressions are being parsed and evaluated by the S executive. Macro time involves the substitution of text into macro definitions, and the generation of text as output. Expressions at this time are subject to the rules of the macro system, not to the rules of S. In particular, expressions are not evaluated numerically or in terms of the contents of datasets referred to. Tests are only on the *text* of arguments.

Section 4 discusses a number of additional topics concerning macros, including: predefined macros, macros with arbitrarily many arguments, conditional expansion, and interaction with the user during expansion.

3.1.2. Applying Functions; Iteration.

Expressions in S operate on whole data structures. Conceptually, a given function acts simultaneously on all the data values in its arguments. To get the full power of the language, it is important to think of computations in terms of the data structures as a whole, in contrast to lower-level programming languages in which computations are on single data values. Various examples throughout the documentation show how computations can be expressed in whole-data-structure terms.

The techniques of this subsection extend the scope of these computations by allowing S functions to be applied over subsets of data and by providing for iterative execution of expressions. These techniques are useful, particularly in conjunction with the macro facility, in developing new analyses. Users who come from a background in lower-level programming languages, however, need to remember that iteration in an interactive, high-level language is intrinsically less efficient than non-iterative use of the language facilities, especially when the iteration is applied to individual data values. With experience, it is often possible to recast the computation into a natural non-iterative form.

The function *apply* allows an arbitrary S function to be applied to sub-arrays of an array, or, more commonly, to the rows or columns of a matrix. The latter cases are most easily handled by calls to the system macros:

[†] Beware, however, of macros which invoke other macros! The intermediate dataset names in all the macros used must be distinct; otherwise, these datasets are likely to be overwritten at the wrong time.


```
?col(x, fun)
?row(x, fun)
```

In both cases, x is a matrix and fun is the name of an S function. The macro $?col(x, fun)$ applies fun to each column of x . If fun returns a result of length n for each column, the result is a matrix with n rows and $ncol(x)$ columns. If n is 1, the result is returned as a vector of length $ncol(x)$.

Similarly, $?row(x, fun)$ returns the result of applying fun to each row of x ; in the form of an $nrow(x)$ by n matrix or a vector. In either case, if fun returns a null result, so does the macro. Additional arguments, if any, are passed directly through to fun . For example,

```
?col(x, mean)
```

returns the vector of column means of x . To generate trimmed column means, by giving an additional argument to *mean*,

```
?col(x, mean, trim=.1)
```

for a trimming argument of .1. By the use of named arguments, it is possible to cause the rows or columns to be associated with any argument to fun .

In the general case of *apply*, the arguments are a multi-way array (see section 3.2.1), a vector *margin* specifying which subscripts should appear in the result, and a character string, whose *value* is the name of the S function to be applied. The following example, on a 3-way array, illustrates this more general form of use. Suppose x is an array of dimension $c(50,4,3)$. We want a 4 by 3 matrix of medians from all combinations of the second and third subscripts.

```
> medns ← apply(x,c(2,3),"median")
```

Here *median* is invoked on 12 sets of 50 numbers each.

The function *sweep* goes along with *apply* in many applications. It takes values produced (usually) by a use of *apply*, and "sweeps" them out of an array. By default, the values are subtracted from the array. System macros *?rsweep* and *?csweep*, analogous to *?row* and *?col*, perform row and column sweeps. To remove column medians from a matrix:

```
> resids ← ?csweep(x, ?col(x,median) )
```

The result of *?col* is a vector of $ncol(x)$ medians; *?csweep* then subtracts each of these from all the values in its corresponding column. Optionally, *sweep* can take the name of an operator or function for the sweep process; for example, "/" to cause the marginal values to be divided into the array. Suppose we want to divide the residuals produced above by the standard deviation of each set. In one expression:

```
> stdized ← ?csweep(resids, sqrt( ?col(resids,var) ),/)
```

As with *apply*, the general form of *sweep*, for use with multi-way arrays, takes a vector of margins and an optional character string to identify the sweeping operation. To sweep out the medians in the 3-way array example of *apply*:

```
> resids ← sweep(x,c(2,3),medns)
```

It is also possible to apply a function to the components of a structure, using the function *sapply*:

```
sapply(z,fname)
```

with z a general structure (see 2.2.5) and $fname$ a character string giving the name of the function (remember to quote it). In this case, the result of the function applied can be any structure. The result of *sapply* will be a structure with the successive results as components. The structure returned will, however, be turned into an n by $ncomp(z)$ matrix if the function always returns n values, and into a vector if n is 1. For example, to find the 10% trimmed means of each of the components of z ,

```
> locn ← sapply(z,"mean",trim=.1)
```

As with *apply*, additional arguments may be passed through to the function being applied. There is a system macro, *?ssweep*, which corresponds to *sweep* for arrays, and has the same arguments as *?csweep*; namely, the data structure from which to sweep values, the marginal values (here the result of *sapply*), and an optional operator for sweeping.

The generation of structures for use with *sapply* often involves splitting up some data of interest by means of a grouping variable. The function *split* accomplishes this, while the function *c* does the opposite job of concatenating the components back into a vector. See section 3.3.2.

The various ways of applying functions over arrays and structures provide an iterative facility suitable to many applications. However, they can only apply a single S function and can not be used if the data has not been organized into a single array or structure. The S language itself has an iterative facility for general situations, although where the previous techniques are applicable, they will be more efficient. The most common form of general iteration is the *for* loop:

```
for( name in expr1) expr2
```

The expression $expr_2$ is executed repeatedly. Before the first execution, $name$ is set to the first data value in $expr_1$, before the second evaluation it is set to the second data value, etc. Execution terminates when there are no more data values in $expr_1$. As an example, suppose we want to regress each of the columns of y against the matrix $xmat$, then plot the residuals on a normal probability plot:

```
for( cy in 1:ncol(y)) qqnorm( reg(xmat,y[,cy])$resid)
```

This will extract successively the columns $y[,cy]$, regress them, extract the *resid* component of the regression and pass this to *qqnorm*. More typically, the expression being iterated is a compound expression. (See section 3.1.3 for a further description of compound expressions). For example, suppose we also want to plot the absolute values of the residuals against the fitted values (y minus the residuals) each time:

```
for( cy in 1:ncol(y)) {
  $Tresid ← reg(xmat,y[,cy])$resid
  $Tlabel ← encode("Regression on column",cy,"of y")
  qqnorm($Tresid,main=$Tlabel,ylab="Residuals")
  plot(y[,cy] - $Tresid,abs($Tresid),xlab="Fitted Values",ylab="Abs. Residuals",
    main=$Tlabel)
}
```

Notice that the example is careful to label each of the plots distinctly. The use of *\$Tresid*, *\$Tlabel* as the names for intermediate datasets follows the practice we recommend for macro definitions. In fact, the user of iterative loops will usually want to put

expressions such as the above into macro definitions. The chance of typing them often without making typing mistakes is not large.

The *name* in a *for* loop is a built-in S dataset. These built-in datasets are quite special. Although they can be referred to in expressions like ordinary datasets, they are not actually stored on a database and their values cannot be modified. When the iteration is complete, they disappear. See section 5.2 for the full definition of built-ins.

3.1.3. Conditional expressions; compound expressions.

Conditional expressions allow the evaluation of expressions to be *conditional* on the logical result of some other expression. The conditional expression is

$$\text{if } (e_1) e_2 \text{ else } e_3$$

Each of the e_i is an arbitrary expression. The first expression is evaluated as a logical vector. If the first element is true, expression e_2 is evaluated and returned as the value of the conditional expression. Otherwise, e_3 is evaluated and becomes the value of the conditional expression. The *else* part of the expression may be omitted; in this case, the whole expression has a null value if the condition is false. If the *else* part is present, it should start on the same line as expression e_2 . Otherwise, S will evaluate the whole expression as an *if* without an *else*.

Notice that this is a sequential process: e_2 and e_3 are not evaluated until the condition is examined, and then only one of them will be evaluated. In contrast, there is a conditional *function*, which operates on its arguments in parallel:

$$\text{ifelse } (e_1, e_2, e_3)$$

This evaluates all of its arguments and returns elements from either the second or third, depending on whether the corresponding element of the first argument is true. Both forms are useful, but they have different effects. The expression

$$\text{if}(\min(y)>0) \log(y) \text{ else } z$$

checks whether the smallest element of y is positive. If so, the logarithm of y is returned; otherwise, the data structure z is returned. Notice that y and z do not need to have similar structure at all. Also, $\log(y)$ is not evaluated at all if any y value is not positive.

On the other hand, one might write

$$\text{ifelse}(y>0, \log(y), 0)$$

This returns a structure like y (because the first argument has this structure) containing the logarithm of all positive y values but with zero wherever y had non-positive elements. However, the logarithm of y is always evaluated, and a warning message is printed if any elements of y are not positive.

Generally, one may want to control more than one expression in a conditional expression or iteration: *compound expressions* allow several expressions to be grouped into one by separating them by semi-colons or new-lines and enclosing the list in braces. For example:

$$\text{if}(\min(y)>0) \{ y \leftarrow \log(y); \text{label} \leftarrow \text{"Log of response"} \}$$

Wherever an expression can be used in S, one can use instead a compound expression (which acts as a single expression). There are three common applications for compound expressions: in conditional expressions, in iterative expressions, and in macros.

The following details about compound expressions are sometimes important:

- The value of a compound is the value of the last subexpression computed.
- The value of an individual expression within a compound expression is never printed automatically; use the *print* function with the expression as argument.

3.1.4. More on Arithmetic and Operators.

In evaluating arithmetic or logical expressions, S attempts to make the result meaningful, using knowledge of the structures of the operands. Note the following specific effects. If two operands are of unequal length, values from the shorter will be reused cyclically, to make the result the same length as the longer. We already saw the most common case of this, when one argument has only a single value, as in $x+1$. Somewhat less obviously:

```
> 1:7 * 1:2
1 4 3 8 5 12 7
```

Whenever a logical vector is used where the context requires numerical values, the logical values are *coerced* to numbers by the definition that TRUE is equivalent to 1 and FALSE to 0. This is true not only for arithmetic, but in any function which expects numbers and gets logical values. As an example, suppose *male* is a logical vector for a set of observations and we want a corresponding vector of labels (perhaps as input to the *text* function). One way to get this is to index a set of two labels with 1 for females and 2 for males:

```
c("Female","Male")[ male+1 ]
```

Conversely, numerical values will be coerced to logical by mapping all zeroes to FALSE and any other values to TRUE.

S functions will not generally coerce from numerical or logical data to character data, or conversely. This can be accomplished by an explicit function or operator; equivalent forms are

```
x %m mode      # operator
coerce(x,mode)  # function
```

The mode provided will usually be one of the built-in modes, REAL, INT, LGL or CHAR. Conversions not involving character data, however, usually are done automatically and do not require the *coerce* function.

When several operators appear in an expression, as in the first example of this subsection, some arbitrariness exists as to which operation is done first. In that example, both occurrences of ":" were evaluated first, and the results then given to "*". This is usually described by saying that ":" has *higher precedence* than "*". Similar questions about how any expressions involving S operators are evaluated may be answered by the following table. Operators higher in the table are evaluated before operators lower in the table. Operators with the same precedence are listed on the same line, and are evaluated left to right.

\$	component select	HIGH
%x	special operator	
-	unary minus	
:	sequence operator	
~	exponentiation	
**		
*	mult/div	
/		
+	add/sub	
-		
< > <= >= == !=	logical	
!	not	
&	and/or	
← - →	assignment	LOW

Remember, when in any doubt, that parentheses can and should be used to make explicit the order of evaluation desired:

```
(1:7)*(1:2)
n:(m-1)
(y-mean(y))*0.1
```

3.1.5. Numbered Components of a Structure.

It is possible, although less frequently useful, to address the components of a structure by position rather than by name; that is, $z[i]$, where i is taken as a single integer value. The main use of this expression is for looping over all the components of a structure (the function `ncomp(z)` gives the number of components).

If you want to apply a function to each of the components of a structure, it is usually preferable to invoke the function `apply` (see section 3.1.2), rather than referring to the numbered components directly. An application of numbered components might be:

```
> z ← split (x, x%/10) # create structure (see 3.2.2)
> for(i in 1:ncomp(z)) {
  stem(z[i]); qqnorm(z[i]) }
```

The first expression splits the elements of a vector according to first digits, and the second does a stem-and-leaf display and normal probability plot of each subset.

3.1.6. Executing Commands on Invoking S.

Users may wish to set options in S or do other operations automatically, whenever they invoke S. This can be done by defining a *profile* macro. When S is invoked, it examines the user's save data base for a macro with the name *profile*. If such a macro exists, it is automatically invoked before prompting the user for command input.

For example, suppose the user wished to set the paper width option to 120 for a wide terminal. Then a profile macro could be defined as follows:

```
MACRO profile
options(width=120)
END
```

Other uses of *profile* might be to include special databases in the search list (section 3.2.3), to invoke a graphics device function, to set other system options by the *options* function, or to attach a chapter of special S functions (section 6).

3.1.7. Batch Execution of S.

There are occasions when interaction with S is unnecessary, e.g., when a large computation is to be carried out utilizing a source file or macro. At these times, *batch* execution of S is desirable since it frees the interactive terminal for other tasks such as editing, etc.

The utility function

S BATCH infile outfile

initiates batch execution of S. Input commands to S come from *infile*, and output is placed on *outfile*. The input file is equivalent to a source file, as described in section 3.1.1. (The *source* function is a good way of interactively testing a file of expressions before using BATCH.) The output file contains a listing of the expressions from *infile* along with the printed output that results from evaluating those expressions. Any datasets created or modified during batch execution are available for use in subsequent S sessions. Also, a deferred graphics file (section 3.4.9) can be produced in batch mode, and later replotted interactively or on a batch device.

3.1.8. Keeping Track of an Analysis: Diary.

In a project that requires an extensive amount of data analysis, it is often difficult to keep track of what procedures have been carried out; it may also be difficult to remember details of how an analysis was performed. In addition, if S is used at a crt (scope) terminal, there is no hard-copy record of the S expressions that carried out the analysis.

The function

diary(on, file)

provides assistance in this area. *Diary* with *on*=TRUE (the default) activates a diary-keeping facility in S which appends the date and time to *file*, followed by all expressions that were typed to S. In addition, any expressions created by macros or source files (section 3.1.1) are also recorded on this file, whose default name is "diary". This provides a record of the actual definition of macros and the content of source files at the time they were used.

Because the diary file contains all input typed to S, it also contains comment lines that the user may type. Thus, thoughts can be recorded by typing comments:

```
> diary # turn diary-keeping on
> plot(x,y)
> # Three outliers appear on the plot -- investigate
```

Also, the lines that came from macros or source files appear on the diary file as indented comments. This makes clear the distinction between typed input and macro or source file input, and means that the diary file itself can be used as a source file to recreate an analysis at some later time.

Diary keeping can be turned off by typing

diary(FALSE)

3.1.9. Functional Form of Operations

Some operations in S can be used in an optional functional form which is usually somewhat more general, but less convenient. These include the functions *get* and *assign* to get and assign datasets, the function *select* to select component of a dataset, and the function *sys* to invoke an operating system command.

The function *get(name)* gets the dataset whose name is the value of the argument *name*. This is the chief advantage of the *get* function; it can be used to access (usually in a *for* loop) datasets from a list of the relevant names. For example, suppose we want a list of the number of values between 40 and 50 in each of the datasets whose names are in a character vector *grades*:

```
count_NULL
for(i in student.grades) {
  count ← c(count, sum(
    get(i)>40 & get(i)<=50 ))
}
```

An alternative approach to the same problem would have been to make all the vectors into components of a single S structure. If *grades* were the structure, then the references to *get(i)* would become *grades\$[i]* and the loop is for *i* in *seq(ncomp(grades))*. When the total size of all the data involved is large, there may be significant differences in efficiency between the two approaches. The use of *get* means that only the data for a single vector is accessed each time in the loop. However, the expression *grades\$[i]* must first access the entire structure, and then select the desired component. In the limit, *grades* could even be too large to access in an S function (obviously, the numerical size at which things start to be a problem depends on the computer used and the response to sizable requests for memory).

A second reason for the use of *get* might be the possibility of constructing dataset names systematically. This tends to go along with the efficiency argument, in that there is not much incentive to construct dataset names unless one is working with a sizable database. Constructed dataset names might be convenient, for example, in setting up a large database using the *extract* utility (section 3.2.4). A third advantage (perhaps) is that names which would not be legal in ordinary S syntax can be used with *get* and *assign*. (This is strongly discouraged in general, but may be needed to get out of some obscure problems.)

The function *get* may take an arbitrary number of names; the effect is to select the first-level component of the dataset from the second name, the second-level component from the third name, and so on. Again, this might possibly be useful if the component names were systematically the same over a collection of datasets. It is also possible to use an argument *pos=* to *get* to restrict the search for the datasets to a single database (see the detailed documentation).

The *assign* function is the symmetric partner to *get*: *assign(name,expression)* assigns the value of *expression* in a dataset whose name is the value of *name*. The applications and relative advantages are similar to *get*; for example, suppose we wanted to use *cut* to make a letter grades out of the numerical grades in the vectors used in the previous example. We will do this by constructing, with the *encode* function, a corresponding vector of names, with "let." on the front of the previous name.

```

for( i in student.grades) {
  assign( encode("let.",i,sep=""),
    cut(get(i),breaks,letters))
}

```

The `sep=""` argument to `encode` prevents internal blanks from appearing in the new names.

It is possible to give an argument `pos=` to `assign` if the assignment should be to a database other than the working data.

The function `select(str, comp1, comp2, ...)` selects the component of `str` defined by `comp1` at the first level, `comp2` at the second level, etc. The arguments for `select` are not as strong as for `get` or `assign`. One may want to use a systematic set of component names, as with `get`. There is then the possible advantage that `str` can be a general expression in the case of `select`.

The function `sys(cmd)` executes the operating system command which is the value of `cmd`. Again, the advantage of the functional form, over using the `!` escape, is the possibility of constricting the command with `encode`. For example, suppose `files` is a character vector with a list of file names. We can construct and execute a sequence of invocations of the operating system command `wc` as follows:

```
sys( encode("wc -c",files) )
```

(in this case, we wanted the blank to separate the command from the file name). The `sys` function executes each element of `cmd` as a separate operating system command.

3.2. Data.

3.2.1. Structures: Arrays; Vector Structures.

The generalization of a matrix is a multi-way array. This is a structure which has a dimension component and a data component. For example, a three-way array with range 1:50 for its first subscript, 1:4 for its second subscript and 1:3 for its third subscript, would be created by the `array` function as follows:

```
> x ← array( read("afile"), c(50,4,3) )
```

The data values on the file are assumed to be ordered so that the first subscript varies fastest, then the second, etc. The first value read from the file becomes $x[1,1,1]$, the second $x[2,1,1]$, and so forth. (See the function `aperm` to rearrange the subscripts of a multi-way array.)

Once created, arrays are used primarily as input to functions like `loglin`, which fits a loglinear model to an array, or to `apply` (see section 3.1.2) which can apply general S functions to subarrays.

Arrays, along with the matrices and time-series introduced earlier, are examples of *vector structures*. A vector structure in S is a structure which can be treated as a vector when suitable, even though it has more structural information than a simple vector. In particular, arithmetic, logical comparisons and numerical calculations can be applied to vector structures. Consider arithmetic or logic; e.g.,

```
x + y;    x >= y
```

First, suppose one of the operands, say `x`, is a vector structure and the other is a simple vector. Then the result is a vector structure with the same structural information

as x but with data determined by the expression. Thus if x is a matrix, $x+1$ is a matrix with the same number of rows and columns, but with data values increased by 1. In nearly all cases, the result is what one would expect intuitively.

When both the operands are vector structures, the situation is more complicated. If both arguments are arrays, then the two dimension vectors must match in all elements, in which case the result is an array of identical dimensions. If the two operands are both time-series, the result is a time-series with time domain the intersection of the two time domains (if possible). The number of observations per period must be the same in both series. For structures other than arrays and time-series, it is necessary that the data components have the same length. Operations failing all of these conditions generate an error.

Most numerical functions operating element-by-element (e.g., *abs*, *sin*) do the obvious: they produce a vector structure with the data values changed and the structure unchanged. However, some functions produce a vector without structure, regardless of the structure associated with the argument. All the sorting functions have this property. For example, sorting the data values in a matrix should not (and does not) produce a matrix.

Occasionally, it is helpful to extract the components of a vector structure. Whatever the type of structure, the component called *Data* is the vector of data values. Thus, in the example of the (50 by 4 by 3) 3-way array, $x\$Data$ is a vector of 600 data values. For a matrix or array $x\$Dim$ is the dimension vector. For a time-series $x\$Tsp$ is the vector of time-series parameters (the starting date, the ending date and the number of observations per unit time).

3.2.2. Prefixes.

It is sometimes useful to make sure that related data sets have similar names. For example, all data generated in a particular analysis may be made to begin with the same initial characters. This can be done automatically in S by specifying a *prefix*; that is, a character string which will then be automatically prefixed to all names used in assignments or references to datasets. For example,

```
prefix("iris.")
```

causes an assignment to the name x to create a dataset on the working database with the name *iris.x*. The use of prefixes reduces confusion between data generated in different analyses, and guards against accidentally replacing a useful dataset that has a simple name (like x).

To refer to a dataset by its exact name, regardless of any prefix in effect, precede the name by "\$":

```
> x ← 1:5; prefix("iris."); x ← 1:2; $x; x
  1 2 3 4 5
  1 2
```

The prefix in effect applies to all the databases currently active; for example, names created in the *save* function will carry the current prefix.

The function *prefix* returns as its value the prefix which was previously in effect. This is useful (particularly in macros) to allow restoration of the existing prefix:

```
oldpref ← prefix("temp")
```

allows one to restore the prefix by *prefix(oldpref)*. The null prefix can be specified

explicitly or by giving *prefix* without an argument.

Another effect of specifying a prefix is that the function *list* will, by default, only list datasets whose names begin with the current prefix. To list all dataset names, it is necessary to give a *pattern* as an argument. The pattern that generates a list of all dataset names is "\$*", e.g.,

```
list("$*").
```

This is a special case of patterns that end in the character "*". Such patterns cause the function to recognize the subset of data names which match the pattern up to the "*". For example, when executing without a prefix,

```
list("iris.*")
```

returns the names of all the datasets on the working database which begin with "iris.". However, patterns in *list* do not have any other special characters except "*" at the end, in contrast to patterns in the text editor, for example.

The function *rm* can be used in an extended form which is also useful in this context. If a special argument *list=v* is given to *rm*, *v* will be taken as a character vector whose elements are the names of datasets to be removed. This form has two advantages. The vector of names can be used to remove the datasets without first having to evaluate them. This is more efficient in the case of many or large datasets. Second, the character string vector can be generated as the value of *list* with a pattern, with the effect of clearing all the datasets with a given prefix.

```
rm(list=list("iris.*"))
```

removes all data whose names start with "iris." This is a powerful solvent: it is usually a good idea to look at the list of names to be removed before removing the data.

3.2.3. Attaching Databases; Search Lists.

When users invoke S, they have available automatically three databases. Other databases may be included explicitly; for example, another user's database or a special database shared by a group of users. To attach a database

```
attach(name)
```

where *name* is the name of the database to be attached. Note that this must completely identify the file. For example, if the database is the save database of another user, the name must be the full name of the user's directory, followed by "/sdata".

The *attach* function puts the new database onto a *search list* used in looking for a dataset. The three original databases plus the databases named in calls to *attach* are searched, in order, for a specific dataset name when the dataset is used in an S expression. By default, *attach* puts the new database onto the end of the search list. It is possible to specify an argument, *pos*, to *attach*. In this case, the new database becomes position *pos* in the search list. Databases that were in position *pos* or beyond are pushed down in the search list. In particular, one may replace the working or save databases by attaching a database with position 1 or 2:

```
> attach("newwork",pos=1)
```

changes the working database to "newwork".

Databases may be detached from the search list once they are no longer needed, by supplying either the name or the position. For example, after attaching the new

working database, one could detach the default working database by either of the following:

```
detach("swork")
detach(pos=2)
```

It is also possible to examine the current search list. The function

```
search()
```

returns the database search list, as a character vector.

3.2.4. Moving Data between Computers; Nonstandard Input.

The data input facilities in section 2.2.3 can read a vector of data values from a file. Additional information about the mode and length of the data is inferred from the data items or is supplied by the user. As the example involving *matrix* in section 2.2.5 illustrated, data structures more complex than a simple vector cannot be read directly in this way. Similarly, there is a *write* function which writes only data values; other information is not part of the output file.

There are, however, other S functions which provide input and output for complete S structures, including the dataset name, mode and length, as well as data values. Although these functions read and write files of character items in human-readable form, they are fully general interfaces for transmitting any S data structures. The applications of these functions are mainly transmitting datasets between computers and providing archival storage, in readable form, of S datasets.

The function

```
dput(x,file)
```

writes the S dataset *x* to the external file, in a self-describing form. If *file* is omitted, the output will be to the user's terminal. Conversely,

```
dget(file)
```

reads an S dataset in self-describing form from the file. For output or input of several datasets, the functions *dump(list,file)* and *restore(file)* may be used. The argument *list* is a character vector whose values are the names of the datasets to be written to the file. Typically, this argument is itself the result of the *list* function, possibly with a pattern argument or with a prefix in effect. The corresponding call to *restore* reads all the datasets from the file and assigns them to a database, using the dataset names read from the file. Either function can take an optional *pos* argument to cause the datasets to be dumped from or restored to the save database or any other attached database. For example,

```
> dump(list(pos=2),"dbfile",pos=2)
```

dumps all the user's save database (or all under the current prefix) to the file "dbfile". The contents of "dbfile" may then be transmitted to another computer on which S runs, either over a data network or, if necessary, through copying to a tape. If the transmitted data were written to a file "newdata" on the new machine,

```
> restore("newdata",pos=2)
```

would restore the data to the save database on the new machine. Note that *dump* and *restore* ignore the current prefix.

User's data files for input may have a more complicated format than the simple "data items separated by white space" that *read* can accommodate. Many of these files can be handled by the S macro *extract*, which is able to deal with character files in which each line (record) represents an observation, and where all lines are in an identical form. Each line can consist of fields that begin and end in fixed character positions (as would be the case with FORTRAN formatted output), or it can contain fields that are delimited by a field-separating character, such as a tab. Both character and numeric fields can be present in each record.

The first step in reading such a file into S is to describe the layout of each record. This description is placed in another file, typically by use of a text editor outside of S. The format of the descriptor file is best shown by example. Suppose a data file named "state" has a state name in character positions 1 to 30 of each line, followed by population in positions 31 to 40, and crime rate in positions 41 to 50. We create a descriptor file named "state.des" that contains:

```
name CHARACTER 1 30
population INTEGER 31 10
crimes REAL 41 10
```

Each line of this file describes one field of the data file, and contains the field name, the mode (CHARACTER, REAL, INTEGER), the starting position in the line, and the length of the field, all separated by white space.

Once a descriptor file is constructed, the *extract* macro can be invoked.

```
> ?extract(state)
```

The macro first invokes a utility function to process the data and descriptor files. The utility function is designed to be usable as a stand-alone utility in the operating system. It is invoked from within the macro by:

```
!S extract state
```

(Thus, the underlying utility is invoked as "S extract" outside of S). A file named "state.ext" is created by the utility. This file is a self-describing character form of a database file, and is then be read into S by

```
restore("state.ext")
```

Thus, after execution of the *restore* macro, three new datasets named *name*, *population*, and *crimes* will be present on the working database.

A similar process is used if the lines of the data file consist of fields separated by a distinctive character. In this case, the descriptor file should begin with a line

```
-f%
```

where "%" is the desired field separator character, and can be omitted if it is a tab. Also, the position numbers should refer to field numbers, and field lengths should be omitted. However, the processing through *extract* is identical.

Several minor points are left. (1) The *extract* procedure does not require the entire file to reside in primary computer memory at any one time. This is particularly important for long files with many fields. (2) Only the positions or fields specified in the descriptor file are processed. The contents of the rest of each record are irrelevant. (3) The descriptor and extract files can have arbitrary names; for details see *extract* in the detailed documentation. (4) It is possible to use the *extract* utility

function on one machine, and then to ship the resulting ".ext" file to another machine for processing by the *restore* function.

In some cases, even the flexibility offered by *extract* may not be sufficient for transferring data to S. In particular, this does not provide access to FORTRAN binary (unformatted) files or data residing in database management systems. For these situations, the data can be written to a file that *extract* can handle.

3.2.5. NULL Data Type.

Occasionally there is a need for a value that represents the absence of any data values, i.e., a *null* value. For example, if there are no data sets on the working database, then the value of the *list* function is NULL:

```
> list()
NULL
```

The value NULL is also useful in a number of situations involving iteration. Suppose, for example, that each iteration of a loop computes some values, and that the user wants a vector of all values computed during the loop. For example, to produce a list of all datasets accessible on any of the first 3 databases,

```
all ← NULL
for (i in 1:3)
  all ← c(all, list(pos=i))
```

In this case, NULL serves as a place holder until some values are produced. (The *c* function throws away NULLs). As always, explicit looping is to be used only if an equivalent computation cannot be carried out by the looping implicit in functions such as *apply* and *sapply*.

3.2.6. Categorical Variables.

The functions *code*, *cut*, and *table* all deal with a data structure known as a *category*. This is a vector structure used to represent categorical or discrete variables. In addition to its *Data* component, it has a component named *Label*. The *Label* component is a character vector naming the various values that the categorical variable can take. The *Data* component consists of integers between 1 and the length of *Label* telling which of the discrete values the variable had for each observation.

Categories are tools for analyzing data which is either qualitative, discrete-valued, or is to be analyzed by breaking quantitative variables into finite ranges. Functions *code* and *cut* form categorical variables from discrete and continuous data values. For example, suppose *occupation* is a character vector with values like "statistician". Then

```
code(occupation)
```

returns a category with *Label* component containing an alphabetical list of the different occupations and *Data* component coding the values as 1, 2, etc.

The *cut* function creates categories from continuous variables. For example, if *income* is a numeric vector,

```
cut(income,4)
```

creates a category with 4 levels by cutting the range of the income values into 4 equal intervals. There are a number of ways to use *code* and *cut* to have control over the coding process; for example,

```
cut(income,pretty(income))
```

uses a set of "pretty" numbers over the range of *income* to define the intervals.

The function *table* generates a contingency table from an arbitrary number of categorical variables (all referring to the same set of observations). A table based on *k* categories gives the counts of the number of observations in the *k*-way array defined by the categories. Specifically, the table is an array with *k* dimensions, the number of levels in the *i*-th dimension being equal to the number of levels in the *i*-th category. In addition the table has a component named *Label* which is made up of all the labels from the categories. The output of *table* is appropriate for analysis by the *loglin* function described in section 3.3.1.1.

Although the normal S *print* function can be used to print tables, they can be printed in a more pleasing form by the function

```
tprint(table1, table2, ... )
```

Unlike *print*, *tprint* prints all tables in parallel, cell by cell. Thus, for example, values of "data", "fit", and "residuals" can be printed together for each cell. See the detailed documentation for more information and a sample printed table.

3.2.7. Documenting Datasets and Macros.

Good documentation can be the key to the usability of macros, and is also important for shared datasets. The function *help* can print documentation for user datasets and macros in the same form as documentation for the system database because associated with each database on the search list is an (optional) collection of documentation. The *help* function searches through these collections whenever macro or dataset documentation is requested. Thus, your macro and dataset documentation is available to anyone who attaches your database.

A macro and several S utilities are available to assist in the creation of this on-line documentation. The first step is to use the system macro

```
?prompt(name)
```

to create a *documentation file*, named "*name.d*". The argument *name* can be the name of a macro, or the name of a dataset or dataset prefix (section 3.2.2). Obviously, *?prompt* cannot write the documentation, but it will make note of macro arguments or dataset names within a prefix. Once the documentation file is created, it should be edited with a text editor so that it adequately describes the macro or dataset. The file will contain instructions (preceded by the character "~") describing what should appear at various places in the file.

Once the documentation file has been edited, it can be installed in the documentation collection by the utility

```
S NEWDOC name.d
```

At this point documentation is accessible via *help*, and the documentation file may be removed.

If the documentation needs to be changed, the documentation file can be retrieved by the utility

```
S EDITDOC name
```

which writes a documentation file "*name.d*". This file can be modified and then re-

installed by the NEWDOC utility.

To provide printed copies of documentation, use the utility

```
S PRINTDOC [options] [name ... ]
```

By default, all documentation is printed at the terminal in the same form as the S detailed documentation. If selected *names* are given, only that documentation is printed. If *options* is "-t", the documentation will be phototypeset; if it is "-o", documentation will be produced on an offline printer.

3.3. Data Analysis.

3.3.1. Statistical Functions.

3.3.1.1. More on Regression and Models.

Section 2.3.1 introduced three functions to fit linear models: *regress*, *lfit* and *rbiwt*. A general routine to fit linear models by robust methods is *rreg*. The simple call is similar to least-squares regression:

```
> z ← rreg(x,y)
```

As with *regress*, *x* is a matrix whose columns are the predictor variables and *y* is a vector to be fitted to the linear model. The model is estimated by an iterated, weighted least-squares fit, in which the weighting attempts to reduce the influence of observations in the data which, on the previous iteration, had unusually large residuals. The method is useful when the usual assumptions of linear modelling have been violated (e.g., because there are a few unusual observations or because the normal distribution is not a good model for the residuals). It is a good practice to compare the robust fit to a least-squares fit from *regress*. If the two are in reasonable agreement as to coefficients and residuals, the use of the least-squares analysis is supported. If there is significant disagreement, the user should study the data and the residuals of the two regressions to understand what is happening. For example, probability plots of the residuals may point out a few unusual observations. The structure returned by *rreg* also includes the final weights, *w*; studying the observations with small weights may shed some light.

There are a number of optional arguments to *rreg* to control the fitting and/or to study the intermediate results. While these may be helpful, their use does require some understanding of how robust regression is done. See the detailed documentation.

A different problem with some linear models is that the number of *x* variables is too large, either practically or because one wants a simpler model for explanatory purposes. The S function

```
leaps(x,y)
```

implements a technique (known as "leaps and bounds") to find *nbest* or more of the best regressions against a *subset* of the *x* variables, according to a criterion of good fit. By default, the criterion is the C_p statistic, measuring the residual mean square against the mean square for the total regression. Other options may be supplied to *leaps*, including other methods of judging the goodness of the fits. Note that the function may return more than *nbest* subsets. The subsets are defined by a component, *which*, of the value of the *leaps* function. This is a logical matrix with as many rows as there

are subsets returned and the same number of columns as *x*. Each row defines a subset of the *x* variables. Other components returned are a vector, *label*, of labels for each of the subsets and a vector giving the values of the criterion for each of the subsets returned. This component has the same name as the method used; e.g., "Cp" by default.

A common use of this function is to produce a C_p plot of the results. This plots the criterion against the size of the subset (which is another component of the value returned by *leaps*). The system macro

```
?Cp(x,y,plot)
```

does this plot and returns the same form of result as *leaps*. If the argument *plot* is FALSE, plotting is suppressed. Other arguments, if supplied, are considered to be arguments to *leaps*.

The *regress* function introduced in section 2.3.1 is actually built up from two related functions: *reg* and *regprt*. The *reg* function actually carries out the regression and returns the regression data structure. This data structure is then given to *regprt* to print the regression summary, and *regprt* then returns the regression data structure with an indication that automatic printing of the result should not occur. For example:

```
r ← regress(x,y)
```

can also be written

```
r ← reg(x,y)
regprt(r)
```

The explicit use of *reg* in the second example has some advantages. If printing via *regprt* is interrupted, the dataset *r* has already been created. Also, it is possible to make use of the results of the regression (e.g., *r\$resid*, or *r\$coef*) prior to printing the results. In addition, there exists a function *regsum* which, when given the regression data structure, produces a result with components containing the statistics that are normally printed by *regprt*. These include the covariance matrix of the coefficients, multiple r-square, etc.

The function *loglin* fits a loglinear model to a table such as the result of the function *table* operating on categorical variables.

```
loglin(table,margins)
```

It is also possible for *table* to be any multi-way array containing counts (≥ 0) of observations in cells. The argument *margins* specifies which marginal totals the loglinear model should fit. It is a vector of integers: subsets of the integers are separated by 0 values. Each subset specifies the factors (i.e., dimensions of *table*) included in one of the marginal totals. For example, with a 3-way table, the expression

```
fit ← loglin(tab,c(1,2,0,1,3,0,2,3))
```

fits all 2-factor interactions.

The function *tprint* can be used to print the results of log-linear fits. Using *fit* above:


```
resid ← tab - fit
flag ← ifelse(abs(resid)>10,"*", "")
tprint(data=tab, fit, residual=resid, flag)
```

prints a table of data, fitted value, and residual, and flags any residuals with magnitude more than 10.

3.3.1.2. Probability and Quantile Functions.

Section 2.3.1 described some random number generators. Corresponding to these are probability and quantile functions. The probability functions take a vector of data values, plus parameters for the distribution, if any, and return a set of probability values (i.e., values of the cumulative distribution function) corresponding to the data values. Each of the probability functions has a name consisting of "p" followed by a code for the distribution family, such as "norm" for the normal. See the list of code names for distributions in Section 2.3.1.

The quantile functions perform the inverse calculation. Given a vector of probability levels, they return corresponding quantile values. Quantile functions are named "q", followed by the name for the distribution. One application of quantile functions is to generate probability plots from arbitrary distributions (see the detailed documentation for the macro `flqqplotfR`).

3.3.1.3. Multivariate Analysis.

A number of S functions produce results useful in analyzing multivariate data. The functions *prcomp*, *cancor* and *discr* perform classical principal components analysis, canonical correlation analysis and discriminant analysis. For principal components

```
prcomp(x)
```

returns the principal component decomposition of the data matrix *x*. Specifically, it finds a rotation of the columns of *x* such that the new first column has the largest sample standard deviation, the second column the largest standard deviation among directions uncorrelated with the first, and so on. The data structure returned has two components: *sdev*, the vector of the standard deviations of the rotated variables, and *rotation*, the orthogonal matrix whose columns are the coefficients for the new variables. (Remember that component names only have to be specified sufficiently to distinguish them: the rotation can be accessed as `prcomp(x)$r`.) If *x* has *p* columns, *sdev* is a vector of length *p* and *rotation* is a *p* by *p* orthogonal matrix. Principal components are sometimes done on correlation scaling; i.e., with each column divided by its standard deviation. The function *scale* performs this scaling:

```
prcomp(scale(x))
```

will produce the desired analysis. Optionally, *prcomp* can also return the data matrix of observations on the rotated variables: see the detailed documentation.

Canonical correlation analysis takes two data matrices and returns a set of transformed variables for each matrix. The first pair of linear combinations have the largest correlation among all linear combinations. The second pair have the largest correlation among variables uncorrelated with the original pair, and so forth. The S function

```
cancor(x,y)
```

returns a structure with three components: *cor*, *xcoef* and *ycoef*. Suppose *x* and *y* have *p* and *q* columns and let $s = \min(p, q)$. Then *xcoef* is a *p* by *s* matrix whose columns are the linear combinations of columns in *x* producing the canonical variables. Similarly, the columns of *ycoef* give the linear combinations of columns of *y*. Both coefficient matrices are scaled so that the sum of squares of each column is one. The component *cor* is the vector of correlations between the canonical variables.

Discriminant analysis in S operates on a data matrix *x*, arranged so that the first rows correspond to observations from the first group, etc.:

```
discr(x,k)
```

The vector *k* gives the group sizes. The first $k[1]$ rows of *x* form the first group of observations, the next $k[2]$ rows the next group, etc. (The macro `?discr(x,group)` allows the rows of *x* to be identified by a grouping vector.) The data structure returned by *discr* has three components: *cor*, *vars* and *groups*. The columns of the matrix *vars* give the coefficients (linear combinations of the columns of *x*) forming the discriminant variables. Each of these discriminates (predicts) a linear combination of the groups. This combination, or contrast, is given by the corresponding column of *groups*. The correlation with which the discriminant variables predict the combination of the groups is given by the vector *cor*. The linear combinations in *vars* are scaled so that the discriminant variables have within-group variance equal to one.

There are several functions which carry out *hierarchical clustering*. This statistical technique operates on a distance matrix which provides all pair-wise distances in a group of objects. If the objects are represented by the rows of a matrix, the function

```
dist(x,metric)
```

computes the distance matrix according to the "euclidean", "manhattan", or "maximum" metric. Then the hierarchical clustering tree is computed from the distances by

```
hclust(dist, method)
```

The *dist* argument can also be computed in ways other than by the *dist* function; for example, it could be the complement of a correlation matrix, $1 - \text{cor}(x, x)$. The value of *hclust* is a *cluster tree structure*: see the detailed documentation for a description of the components of the structure.

Other functions use the cluster tree structure to produce further information. To produce an assignment of the *n* objects into *k* clusters, use

```
cutree(tree,k).
```

It is also possible to cut according to the height, *h*, at which hierarchical clusters were merged (see the detailed documentation). Given a set of objects named *leaves* i.e., a vector of integers in the range 1 to *n*,

```
subtree(tree,leaves)
```

returns the subtree which contains all the objects specified. For example,

```
subtree(tree,c(1,2,10))
```

returns the tree which contains observations 1, 2 and 10 in the original data. Note that the numbering in the subtree remains that used in the original tree, so that the identification of objects is correct for the original data.

The function *mdscale* performs a version of metric multidimensional scaling; given a set of dissimilarities (distances) among some set of objects, it attempts to construct a set of observations which reproduce these distances as nearly as possible.

3.3.1.4. Matrix Methods; Linear Algebra.

For users with some familiarity with numerical matrix methods and numerical linear algebra, S provides a number of functions which carry out major operations in this area. The S functions can be combined (frequently through macros) to generate customized numerical or statistical procedures. Matrix products and cross products are produced by the operators:

```
x %* y
x %c y
```

The *%c* operator is equivalent to $t(x) \%* y$. The *%o* operator forms outer products. For example, if x and y are vectors, then $x \%o y$ is a matrix whose $[i,j]$ element is $x[i] * y[j]$. It is possible to use *%o* with arrays (see the detailed documentation) and to combine elements with an operation other than *"*"* (see *outer* in the detailed documentation).

Inversion of matrices and the solution of systems of linear equations are handled by the function *solve*. With one argument, *solve(x)* inverts the square matrix x . With two arguments, $y \leftarrow \text{solve}(A,b)$ solves the system of linear equations $A \%* y = b$.

Matrix decompositions are available: *eigen(x)* and *svd(x)* return structures representing the eigenvalue and singular-value decompositions respectively. The function *gs(x)* returns the orthogonal decomposition of the rectangular matrix in the iterated Gram-Schmidt form (i.e., as a matrix of orthonormal columns, q , and an upper-triangular matrix, r). The function *chol(ss)* returns the upper-triangular factor for the Choleski decomposition of the (square, positive-definite) matrix ss . With either of these decompositions, one often needs subsequently to solve triangular systems of equations: this is done by the function *backsolve(R,b)*, which returns the solution of the upper-triangular system of equations $R \%* y = b$ (the optional argument *lower=T* solves lower-triangular systems).

3.3.2. Data Manipulation.

Some functions are particularly useful for handling hierarchical structures. The function *split* splits the elements of a vector to form the components of a structure:

```
split(x,group)
```

takes a vector x and a corresponding vector *group*. Each distinct data value occurring in *group* generates a component in the structure returned by *split*. This component consists of all the data values in x corresponding to the specific value in *group*. For example, suppose *income* and *age* are parallel vectors giving income and age for a set of people. We can generate a structure with one component containing all the incomes for each 10 year range of ages by:

```
> z ← split(income, age%/10 )
```

The result of *age%/10* is a vector with 0 for ages 0-9, 1 for ages 10-19, etc. Then *split* will put all elements of *income* for which the corresponding age is in 0-9 into the first component of z , etc. The values of *group* may be of any mode.

The opposite process to *split* is to take data from a structure and combine it into a single vector. This is accomplished by the same combine function, *c*, used to put together data values from vectors. In the example, *c(z)* would put the income data back together (but not necessarily in the original order).

The function *ncomp(z)* returns the number of components of the structure *z*, while *compname(z)* returns a character vector containing the names of the components.

It is possible to put together and modify hierarchical structures "manually", perhaps as preliminary to a function like *boxplot* or because the result of *split* is not quite as desired. The functions *cstr* and *mstr* create hierarchical structures and modify existing structures:

```
cstr(arg1,arg2, ... )
```

creates a structure whose first component is the value of *arg1*, whose second component is the value of *arg2*, etc. If the arguments are given in the *name=value* form, *name* becomes the component name. To modify an existing structure:

```
mstr(z,arg1,arg2, ... )
```

The component of structure *z* with the same name as *arg1* is replaced by the value of *arg1* and so forth. If no component of this name exists, the argument is added, as in *cstr*. A component may be deleted by giving an empty named argument:

```
> a ← mstr( tbl, labels= )
```

deletes the component *labels* from *tbl*.

The function *encode* creates a character vector out of any combination of character vectors and numeric vectors. These are frequently useful as labels for printing or plotting.

```
print(x,collab=encode("Election of",seq(1912,1980,4)))
```

The corresponding elements of all the arguments to *encode* are concatenated to form a single character string. The vector returned has as many character strings as the longest argument; shorter arguments are reused cyclically (as with the first argument in the example). Arguments which are not character strings are converted in *encode*. By default, a single blank is inserted between values from successive arguments. The special argument *sep=string* allows the user to specify any separating string desired; most usefully, *sep=""* jams successive arguments together without blanks.

3.4. Graphics.

3.4.1. More High-Level Functions.

Section 2.4 introduced functions *plot*, *matplot*, *qqplot*, *qqnorm*, *tsplot* and *hist* to produce graphical displays. Further high-level graphical display functions are introduced here.

A *box plot* is a graphical display to summarize the shape of several sets of data:

```
boxplot(x1, x2, ... )
```

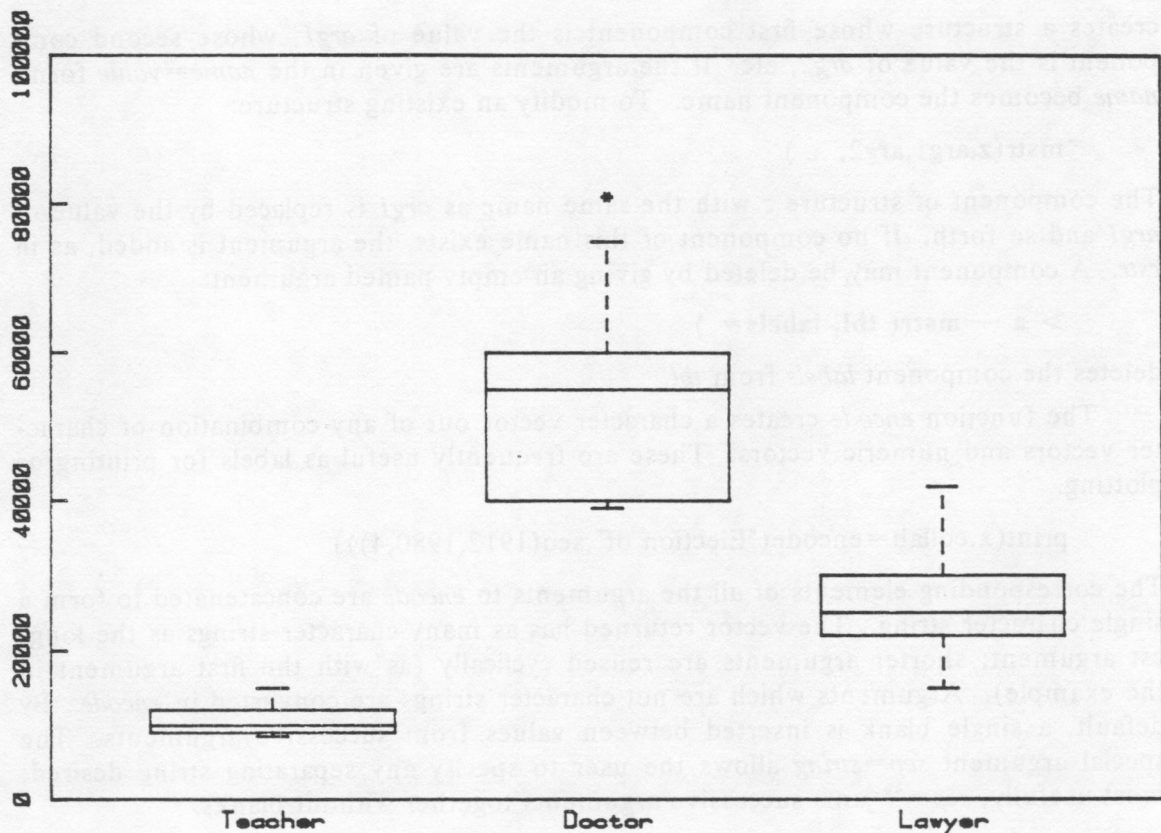
plots each *x_i* in the form of a symbolic, vertical box. The lower edge of the box is at the first quartile of the data, the upper edge at the third quartile, and a line through the box indicates the median of the data. Vertical "whiskers" go out of the box above and below, to indicate the range of the data. However, if points lie outside a robust

estimate of the range, these points are plotted separately.

The *boxplot* function can also take arguments that are hierarchical data structures composed of a number of vectors. These structures are most often created by the *split* function (section 3.3.2). In this case, each component of the structure is represented by a box, which is labelled by the corresponding component-name. For example,

```
> boxplot(split(income,occupation))
```

could produce the plot below. There also exists a more specialized box plot function *bxp* that takes a matrix that describes the positions of box edges, medians, and whiskers. See the detailed documentation.

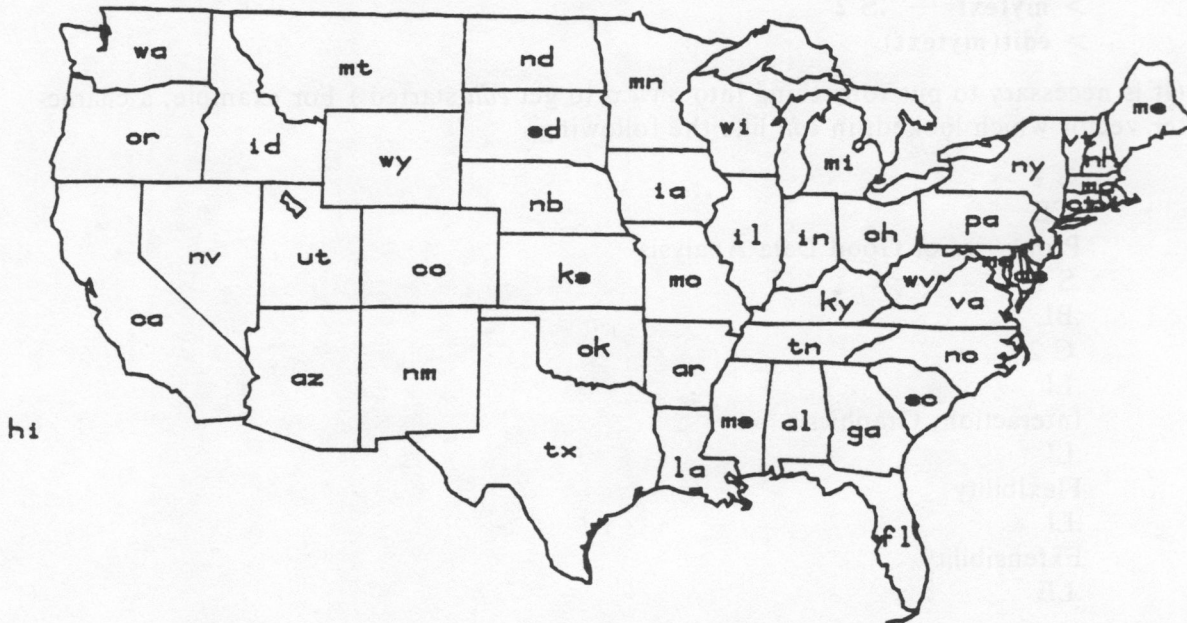


A rather specialized high-level plotting function is *usa*, which produces maps of the United States. Just typing *usa()* gives a map of the coast line and state boundaries. The argument *states=FALSE* will omit the state boundaries. In addition, it is possible to specify a vector giving lower and upper limits for latitude (*ylim*) and longitude (*xlim*) to get a rectangular section of the map, e.g.,

```
usa(xlim=c(65,100),ylim=c(35,50))
```

for the Northeast. Unfortunately, the resolution of the map does not increase as smaller regions are displayed, so maps of small areas are likely to look coarse. The coordinate system for the map places latitude on the y-axis and *negative* longitude on the x-axis (so that coordinates increase from left to right). The system database contains (under the prefix "state.") several datasets with coordinates appropriate for the map. For example, the dataset *state.center* gives coordinates near the center of each

```
usa; text(state.center,state.abb)
```



```
barplot(height, width, names)
```

A simple plot that describes the breakdown of a whole into parts is the *pie chart*. The function

`pie(x, label)`

The function *plotfit* is another high-level plotting function, which is designed to display the results of a two-way analysis of a data matrix. The rows and columns can be labelled, and residuals can be displayed with or without the fitted grid. For details and a sample plot, see the detailed documentation.

S can be used to generate layouts of text information for use in vu-graphs (transparencies), slides, etc. The function *vu(text)* takes a character string and plots the individual character strings as lines of text. Special character strings are interpreted as layout commands, character-size changes (.S), color changes (.C) etc. in a style analogous to *nroff/troff* macro systems. Generally, the best way to create or modify a vector for use with *vu* is via the *edit* function:

```
> mytext ← ".S 2"
> edit(mytext)
```

(It is necessary to put something into *mytext* to get *edit* started.) For example, a character vector which looked, in *edit* like the following:

```
.S 2
.CE
Properties of Good Data Analysis
.S 1
.BL
.C 2
.LI
Interaction, Graphics
.LI
Flexibility
.LI
Extensibility
.LE
```

produces a centered title and a bullet list (.BL) of three items. The title is twice the size of the list items and in the standard color (color 1). The list items are in color 2. The actual size of the characters is chosen by *vu* to fill the plotting surface to the width and height desired (7 inches square by default). See the detailed documentation for other facilities of *vu*.

An additional q-q plot function is *qqgamma(xy,par1)*. Given some data in *xy* and a shape parameter *par1*, the function will produce a probability (q-q) plot against the corresponding gamma distribution. It is possible to do a q-q plot against an arbitrary distribution (provided all parameters other than location and scale are specified). See *qqplot* in the detailed documentation of the system macros.

3.4.2. Contour and Perspective Plots of Surfaces.

Several S functions are concerned with the display of surfaces defined as a function of two variables. The first,

```
contour(x, y, z, v)
```

produces a *contour plot* of the surface which shows lines of constant height. The vectors *x* and *y* give positions along the x- and y-axes and the matrix *z* gives the height of the surface at the corresponding point. That is, *z[i,j]* is the surface height at *(x[i],y[j])*. Contours are drawn at the heights given by *v*, and if *v* is omitted, approximately 5 equally spaced contours are drawn.

The function

```
persp(z, eye)
```

provides a 3-dimensional perspective view of a surface with hidden line removal. The

matrix z is assumed to have come from an underlying regular grid of equally spaced x and y values. It is also assumed to be centered at the origin and to extend ± 1 in the x and y directions. The vector *eye* gives the (x,y,z) coordinates of the viewpoint in this coordinate system. The default viewpoint is $(-6, -8, 5)$. Because of the implied coordinate system and because the surface is considered as a 3-dimensional object with comparable x -, y -, and z -coordinates, values in z should ordinarily be less than one.

The function

```
interp(x, y, z)
```

provides a facility for creating the grid of surface heights needed by *persp* and *contour* from irregularly spaced (x,y,z) data points. The surface is approximated by triangular patches joining groups of 3 data points, and the regular grid of surface values is interpolated. The output structure contains components x , y , and z . The first two are vectors defining the x - y grid, and the last is the matrix of interpolated values. The functions *contour* and *persp* recognize this data structure in place of their corresponding arguments. Suppose, for example, that we have pollution measurements at various positions of latitude and longitude. To produce a contour plot of this data on a map of the region,

```
rx ← range(longitude)
ry ← range(latitude)
usa( xlim=rx, ylim=ry, lty=2 )
contour( interp(longitude,latitude,pollution), add=TRUE )
```

The result is shown below. Contours will not be drawn outside the convex hull of the data, since *interp* produces NAs rather than extrapolating.

3.4.3. Multivariate Plotting Functions.

The following functions implement several methods for displaying the data in a data matrix with several columns. They attempt, so far as possible, to find the structure in multivariate data that can be detected by use of scatter plots in bivariate data.

The simplest idea of this form is to plot all the pairwise scatter plots; i.e., to produce a scatter plot of each pair of the columns of the original data:

```
pairs(x,labels)
```

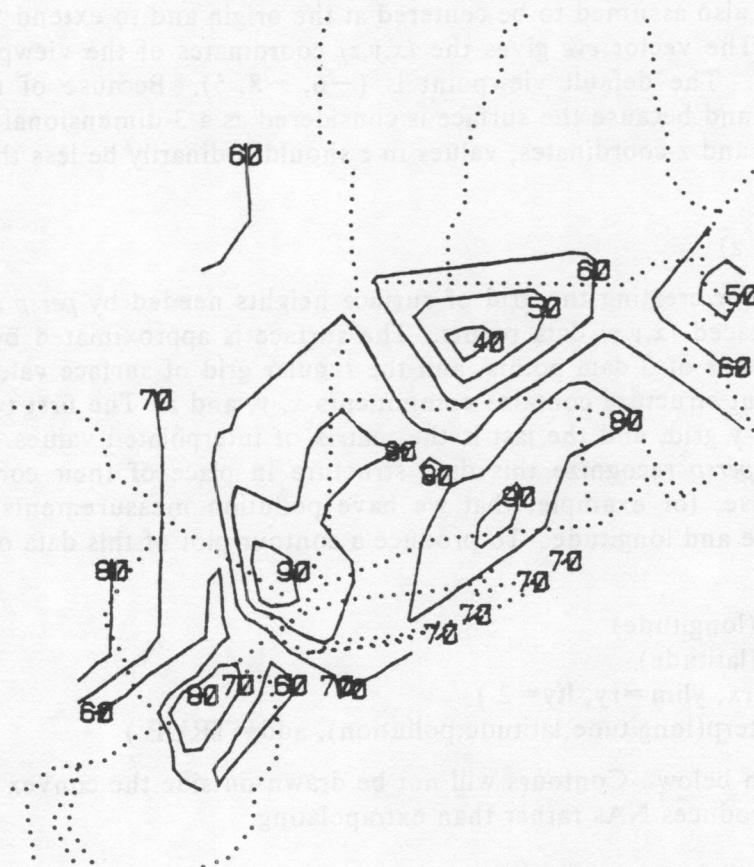
produces on one page all the pairwise scatter plots of columns of the matrix x . The plots are put into a square array of $p-1$ rows and columns, where p is $ncol(x)$. The rows correspond to the first $p-1$ variables (ordered down from the top) and the columns to the last $p-1$ variables. The optional argument, *labels*, is a character vector of labels for the p variables. Other optional arguments supply a main title and the maximum size of the array of plots on one page. See the detailed documentation. The *pairs* function is intended for pairwise scatter plots of a fairly large number of variables (5 to 10 or more). It plots the scatter plots close together, and omits axis scales and labels. For plots of fewer variables, one may use the *?pairs* macro.

Other techniques for plotting multivariate data produce a graphical symbol for each of the rows (observations) in x :

```
stars(x)
```

```
faces(x,which)
```

The *stars* function plots a star or polygon for each row of x . This is a symbol consisting of p radii, the length of the radius being proportional to the value $x[i,j]$ for the j -th



radius of the i -th symbol. Additional arguments allow variations in the form of the symbol. The *faces* plot operates similarly, except that the data values are used as parameters to control various features of a cartoon face. A maximum of 15 variables may be plotted with faces. For both functions, optional arguments provide headings for the plot, labels for each symbol and control scaling. By default, each column of x is scaled linearly to the range 0 to 1 before plotting. If the user wishes to override such scaling (for example, because all the columns are in the same units and can be compared), the argument *scale=FALSE* may be given. In this case, the user should arrange that the data be rescaled overall to the range 0 to 1 before calling *stars* or *faces*. See the detailed documentation for these optional arguments. See also the detailed documentation for the function *starsymb*, which can be used to plot a single star symbol, labelled with the names of the variables (columns) of the data.

A semi-graphical function for displaying multivariate data is

```
smatrix(x)
```

This prints a matrix of characters representing the data values in x . Larger values are represented by "darker" characters. Overstriking can be used for each character position to obtain a richer set of darkness levels. Optional arguments control the choice of characters and the number of overstrikes. See the detailed documentation. The symbolic matrix is a relatively crude plot, but is able to represent much larger datasets effectively than most of the methods mentioned previously.

The cluster trees produced by the *hclust* function are also used to represent relationships within multivariate data. The function

```
plclust(tree, label)
```

produces a plot called a *dendrogram* or a *cluster tree* of the structure produced by *hclust*. The argument *label* can be used to provide appropriate labels for the objects in the tree. A variety of optional arguments allow: leaves which hang a fixed distance down from the nodes or which drop to the bottom of the plot; nodes to have square or angled branches; or squashed or cut-off trees. Another function *labclust* provides more control over the labelling of dendrograms, and is described fully in the detailed documentation.

3.4.4. Graphical Data Structures; Missing Values.

A common feature of many graphical functions in S is the presence of arguments *x* and *y*. Because these arguments are often used together, a *graphical data structure* that combines the two is available. This is a hierarchical structure with components named *x* and *y* that are vectors of equal length containing a set of (*x*,*y*) pairs. Most functions that expect two arguments *x* and *y* can, instead, receive a single graphical data structure. The usefulness of the graphical data structure is that it may be generated as the result of an S function. For example, the function *lowess* is a locally-weighted scatter plot smoothing procedure. Its input is a set of coordinates locating the points on a scatter plot, and its output is a set of coordinates describing a smooth curve through the center of the scatter plot. Thus, *lowess* returns a graphic data structure that can be processed by other functions, e.g.,

```
plot(x,y)
lines( lowess(x,y) )           # add smooth curve
```

Other functions returning graphical data structures include *density* (smoothed probability density estimates) and *rdpen* (graphical input).

Some high-level plotting functions can be used with an argument *plot=FALSE*. In this form, they do no plotting, but instead return a data structure (such as the x-y structure above) which may be used subsequently as an argument to other functions. For example, *qqplot* and *qqnorm* will return an x-y structure representing the ordered quantiles and data values which would have been plotted. The *identify* function returns the indices in the arguments corresponding to the identified points. If *boxplot* is called with *plot=FALSE*, the structure returned is a matrix giving 5 statistics for each of the data sets given as arguments. The function *hist* will return a data structure suitable as an argument to the *barplot* function. The function *plclust* returns a structure suitable for an argument to the function *labclust*.

Many of the plotting functions try to make sense of missing values (NAs) in the data. The general philosophy is that no point will be plotted if one or more of its coordinates is NA, and no line segment will be drawn if either of the end points is NA. This applies to *plot* and other scatter-plotting routines, and to the lower-level functions *points*, *lines* and *text*. The same approach is used by the *contour* function: no segment of a contour line will be drawn to an edge if either of the vertices of the edge is NA in the *z* matrix. An example of this was shown in section 4.4.1 above.

3.4.5. Graphical Input; Identify, Rdpn.

One of the advantages of interactive data analysis lies in the ability to carry out the analysis in a sequential manner, using previous results to influence the course of the later analysis. One area where this is particularly effective is graphics. Often, the eye is the most efficient detector of interesting points on a scatter plot. S provides a *graphical input* facility to capitalize on this.

If a set of (x,y) values have been plotted, say by `plot(x,y)`, then

```
identify(x, y, label)
```

allows the user to "point at"[†] interesting points on the plot, and places the corresponding character-string from *label* beside the identified point.

For example, suppose we are analyzing data given for the states of the US in the system shared data base. The matrix `state.x77` contains per-capita income and life expectancy for the year 1977 in columns 2 and 4. The expressions

```
> prefix("state.")
> list(pos=3) # whats on system db under "state."?
  "abb"  "center" "col77" "name"  "x77"
> # we remember x77 is a matrix of state data
> col77 # names of columns of x77 matrix
  "Pop. (1000s)" "Income"      "Illiteracy"  "Life Exp."
[5] "Murder/1000"  "H.S. Grad.%" "Days of Frost" "Area"
> income ← x77[,2]
> life ← x77[,4]
> plot(income,life)
> identify(income,life,abb)
```

produce the plot, read the position of the terminal's graphic input device, and label the identified points by state abbreviation, as shown below.

The function

```
rdpn(n)
```

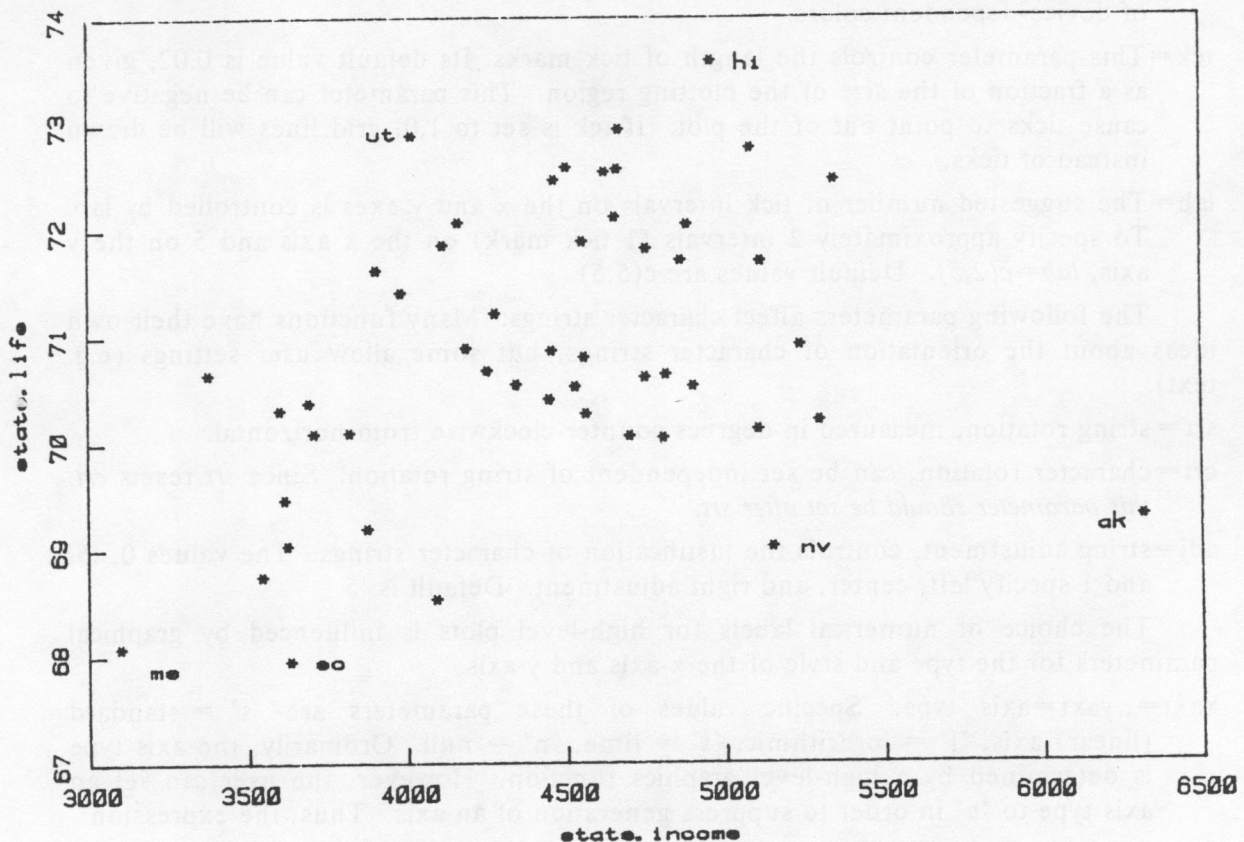
reads up to *n* points using the terminals graphic input facility, and returns their values as a graphical data structure (section 3.4.4). See also the example in section 3.4.8.

3.4.6. Graphical Parameters.

Since there are a large number of graphical displays useful for data analysis, S cannot provide a high-level function for every one of them. Instead, S provides a number of basic high-level plotting routines, some lower level functions to augment plots, and a set of graphical parameters. These are the building blocks for constructing many different kinds of plots.

This section will describe a number of graphical parameters that are useful in constructing special-purpose plots. Section 3.4.7 introduces graphical parameters that control the arrangement of plots on the page, and section 3.4.8 describes functions that can be used in building plots.

[†] The mechanism for "pointing" differs among different graphic devices. Pen plotters normally allow manual positioning of the pen; scopes normally have a cursor. See the detailed documentation of the graphic device.



Graphical parameters, introduced in section 2.4, may be used as arguments to any graphics functions to control *temporary* parameter settings for color, line type, plotting character, etc. These parameters should be given in the *name=value* form. The parameter settings stay in effect through the graphic function and are then reset to their previous value. The following are the more useful parameters:

pch=plotting character. The character given is used for scatter plots, in calls to *points* and similar functions. For example,

```
plot(x,y,pch="+")
```

produces a scatter plot with the character "+" at each of the (x,y) points.

cex=character expansion, the value tells how much to expand (or shrink) the characters relative to standard size for the graphical device. For example, *cex*=0.5 gives half-size characters, *cex*=1.5 gives characters 1.5 times normal size.

lty=line type. For devices that support multiple line-types, values 2, 3, ... will select a non-solid line type.

las=label style. If *las*=1, all axis labels will be horizontal. By default (*las*=0), they are oriented parallel to the axis on devices that have the ability to rotate characters.

`col=color`. Devices which can automatically change color will change to one of a set of device-dependent colors.

`tck`=This parameter controls the length of tick marks. Its default value is 0.02, given as a fraction of the size of the plotting region. This parameter can be negative to cause ticks to point out of the plot. If `tck` is set to 1.0, grid lines will be drawn instead of ticks.

`lab`=The suggested number of tick intervals on the x and y axes is controlled by `lab`. To specify approximately 2 intervals (1 tick mark) on the x axis and 5 on the y axis, `lab=c(2,5)`. Default values are `c(5,5)`.

The following parameters affect character strings. Many functions have their own ideas about the orientation of character strings, but some allow user settings (e.g. `text`).

`srt`=string rotation, measured in degrees counter-clockwise from horizontal.

`crt`=character rotation, can be set independent of string rotation. Since `srt` resets `crt`, *this parameter should be set after srt*.

`adj`=string adjustment, controls the justification of character strings. The values 0, .5, and 1 specify left, center, and right adjustment. Default is .5.

The choice of numerical labels for high-level plots is influenced by graphical parameters for the type and style of the x-axis and y-axis.

`xaxt=`, `yaxt=`axis type. Specific values of these parameters are "s" = standard (linear) axis, "l" = logarithmic, "t" = time, "n" = null. Ordinarily, the axis type is determined by a high-level graphics function. However, the user can set an axis type to "n" in order to suppress generation of an axis. Thus, the expression

```
plot(x,y,xaxt="n")
```

produces a scatter plot, but does not generate an x-axis with ticks or labels.

`xaxs=`, `yaxs=`axis style. Values are "s" = generate an axis with "pretty" endpoints and tick intervals that contains the range of the data values, "e" = extend the standard axis so that points at the limits are not exactly on the axis boundaries, "i" = internal axis where all tick marks are inside the range of the data, "d" = direct axis. The direct axis style allows an axis to be "locked in" so that it cannot be changed by following high-level plotting functions. For example

```
plot(x,y,xlim=c(0,1),ylim=c(0,100)) # set up special axes
par(xaxs="d",yaxs="d") # lock the axes in
plot(x2,y2) # plot will have same axes as previous plot
```

will cause a special set of axes to be propagated to future plots. Remember, another use of the `par` function is needed to "unlock" a direct axis, i.e.,

```
par(xaxs="s",yaxs="s") # set back to standard style
```

Graphical parameters may be set permanently by the function `par`. Parameters set in a call to `par`, stay in effect throughout all plotting until reset by another call to `par` or set back to their default values when a graphic device functions is invoked. For example,

```
par(pch="+",cex=.5)
```

sets the plotting character and reduces the character size for all graphical functions which follow. The layout parameters of section 3.4.7 can *only* be set in a call to `par`,

since they affect all subsequent plots.

A number of special arguments are recognized by many of the plotting functions. `xlab=`, `ylab=` labels for the x or y axis. These labels default to the names of the data sets plotted on the x or y axis.

`main=` a main title plotted in enlarged characters above the plot.

`sub=` a sub-title for the bottom of the plot, below the x-axis label.

`xlim=`, `ylim=` limits (a vector with two values, the lower and upper limits) for the x or y axis. For example, `xlim=c(0,100)`.

`type=` the type of plot: "p" (points), "l" (lines), or "b" (both). Also "n" (nothing) sets up the axes, etc., but does not plot points, and "h" provides high-density vertical lines.

Several functions provide extended interpretations of some of the graphical parameters. In particular, the functions *matplot* and *tsplot* allow the parameters `lty=`, and `col=` to be vectors, which are cycled through for each successive set of data. Also, parameters `pch=` and `type=` are character strings where each character in turn is taken as the parameter value for successive sets of data. Thus,

```
tsplot(hstart, smooth(hstart), type="pl", col=c(1,3))
```

plots the series *hstart* as a set of points in color 1, and adds the series *smooth(hstart)* as a set of line segments in color 3.

To generate a printout of some or all of the graphical parameters, use the function *pardump*. With no arguments, *pardump* prints out the names and current values of all graphical parameters. Optionally, it can be given the names of the parameters of interest, in which case only these parameters will be supplied:

```
pardump("pch","usr")
```

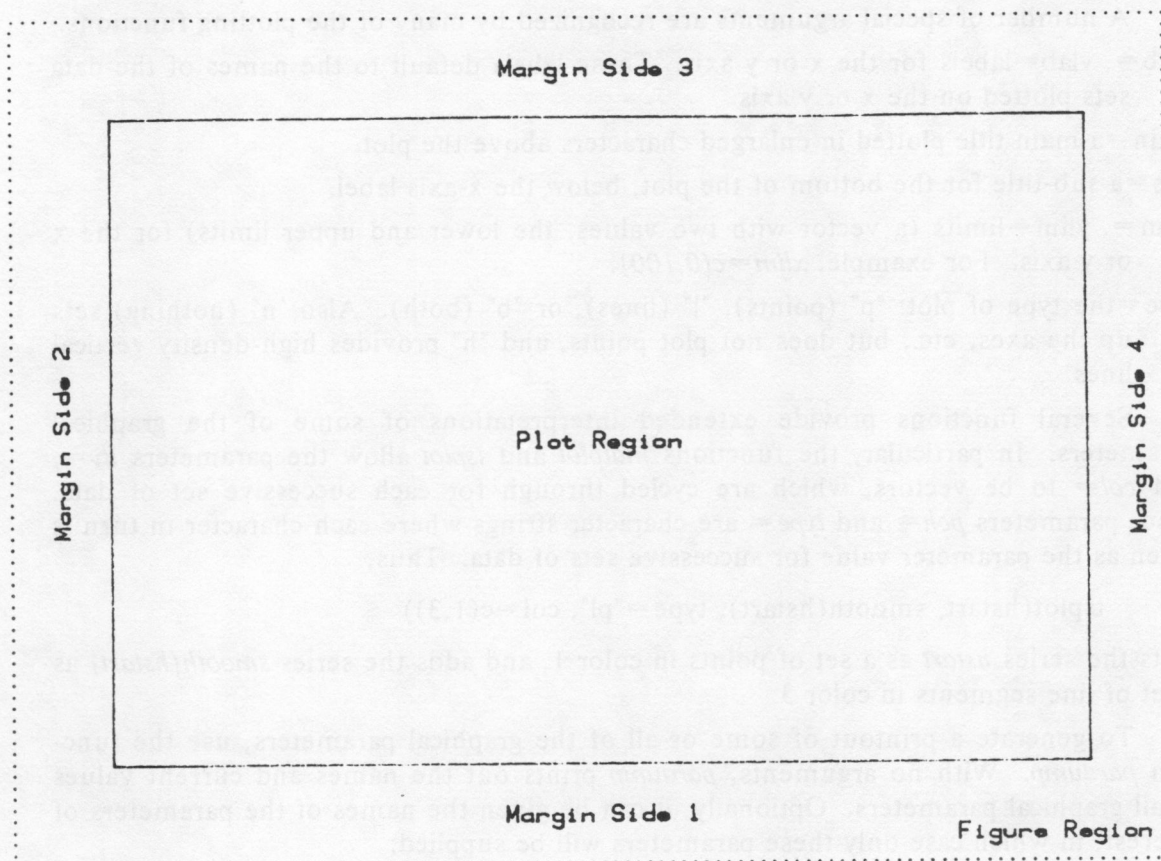
will print out the current plotting character and the range of user coordinates. Since the total printout takes about 600 lines, supplying arguments is a good idea in interactive use.

The values used by the graphics functions in labelling axes can be generated explicitly by the function *pretty(x,nint)* which returns approximately *nint* values (5 by default) that cover the range of values in *x* and are "pretty" in the sense that their difference is a simple decimal number. These values are useful, for example, as contour levels to the *contour* function.

3.4.7. The Layout of Plots and Figures.

Further description of S graphics will be helped by the introduction of some terminology. A simple graphical display including titles and axes is called a *figure*. A figure is composed of a *plot* surrounded by *margins* as shown below. The plot contains the graphical information (the plotted points of a scatter plot, the bars of a histogram, etc.) and is addressed by a *user coordinate* system; that is, by the co-ordinates determined by the data. The margins surrounding the plot are used for axis labels and titles. Locations within the margins are specified by *side* (1 = bottom, 2 = left, 3 = top and 4 = right) and the number of lines of text away from the edge of the plot.

A more complicated situation involves the positioning of an array of figures on a single frame of output as shown below. In this case, each figure contains its own plot



and margins, and the entire page has a set of *outer margins* surrounding the figures.

The function *par* provides control over all graphical parameters that deal with the size and layout of plots, margins, and figures. As noted in the previous section, it can also be used to set other graphical parameters in effect permanently; i.e., until the device is re-specified or the S session is ended.

A frequently useful parameter is *mar=* which controls the size of the margins which surround the plot. The margins by default are set large enough to accommodate 5.1, 4.1, 4.1, and 2.1 lines of text on the bottom, left, top, and right of the plot. The parameter *mar* specifies the size of the four margins, e.g.,

```
par(mar=c(6,6,6,6))
```

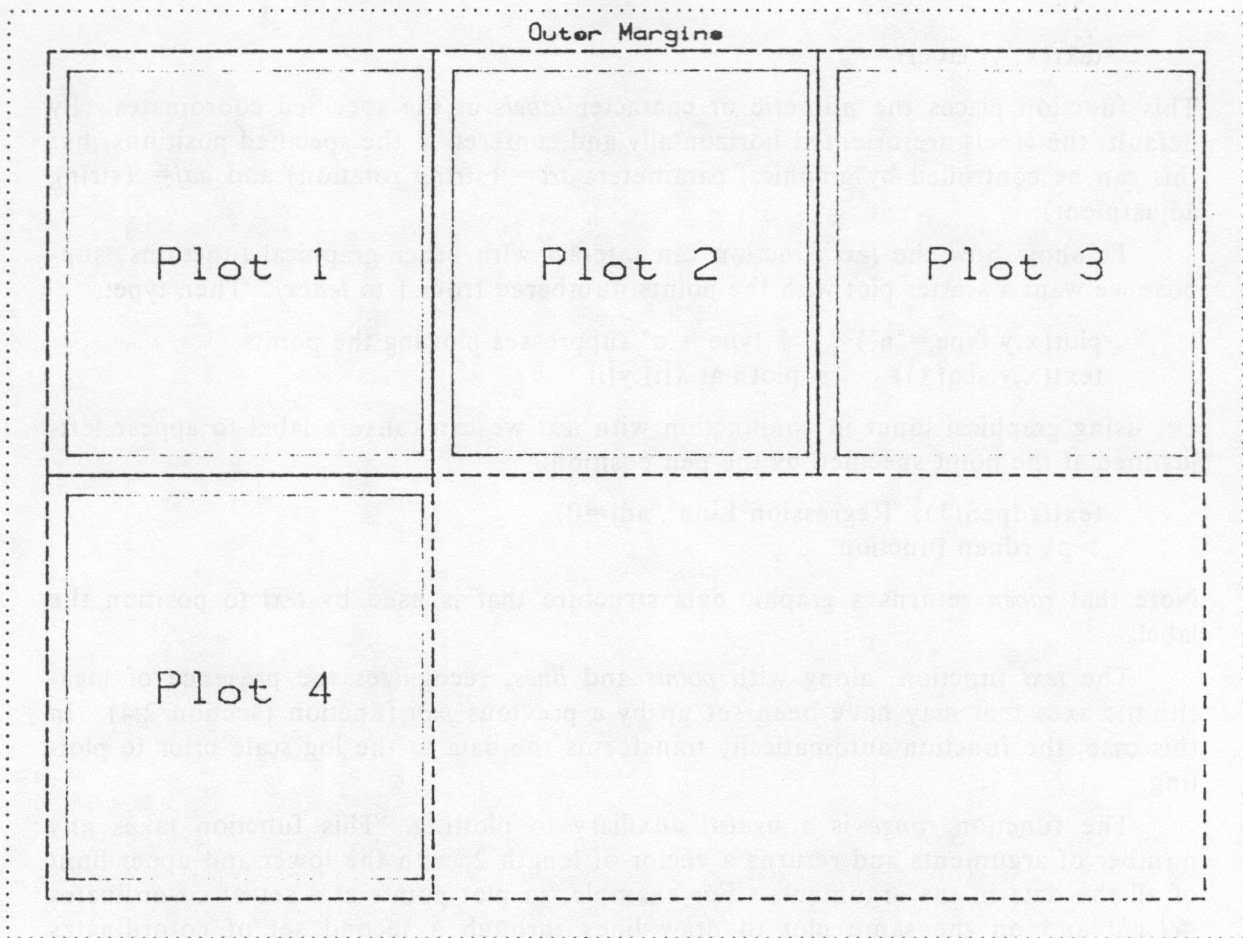
sets up all four margins to accommodate 6 lines of text. A similar parameter *oma=* specifies the size of the outer margins. By default, there are no outer margins.

The parameter *pty* is a character string which describes the plot shape. By default, it is "m" which sets up the maximum size plot possible on the device, after allowing for the margins. The other possibility is "s" which will force the plot to be square, regardless of the shape of the device.

Graphical parameters also provide the ability to set up an array of figures on an output device. To have 6 plots appear on a page (3 rows of 2 plots each), use

```
par(mfrow=c(3,2))
```

Each successive high-level plotting function will automatically advance to the next



available figure region on the page, in a row-first order (first the top row, first column, then the second row, first column, and so forth). When a page is exhausted, a new one is started, still in the 3 by 2 configuration. Similarly, the graphical parameter *mfcol* sets up an array of figures and accesses them in a column-first order. In addition *mfrow* and *mfcol* automatically reduce character and margin size to half the normal size if there are more than two rows or columns.

Plotting can be returned to the normal one-per-page form by

```
par(mfrow=c(1,1))
```

This also restores normal character size.

In general, all high-level plotting routines first "set up" a plot by specifying appropriate user coordinates prior to actual plotting. In some circumstances, it may be reasonable to set up the user coordinates without using a high-level plotting routine. This can be carried out by using the parameter *usr*=(*xl,xu,yl,yu*) where *xl* and *xu* are the lower and upper limits for the x-coordinates, and *yl* and *yu* give the range of the y-coordinates.

3.4.8. Building-up Plots.

The graphical parameters introduced in the previous two sections are typically used in conjunction with lower level graphical functions to build up special plots. One of the most useful of these lower level functions is


```
text(x, y, label)
```

This function places the numeric or character *labels* at the specified coordinates. By default, the labels are oriented horizontally and centered at the specified positions, but this can be controlled by graphical parameters *srt*= (string rotation) and *adj*= (string adjustment).

To show how the *text* function can interact with other graphical functions, suppose we want a scatter plot with the points numbered from 1 to *len(x)*. Then type:

```
plot(x,y,type="n")    # type="n" suppresses plotting the points
text(x,y,seq(x))      # plot i at x[i],y[i]
```

Or, using graphical input in conjunction with *text* we can cause a label to appear left-justified at the point specified by the pen position:

```
text(rdpen(1), "Regression Line", adj=0)
>px rdpen function
```

Note that *rdpen* returns a graphic data structure that is used by *text* to position the label.

The *text* function, along with *points* and *lines*, recognizes the presence of logarithmic axes that may have been set up by a previous *plot* function (section 2.4). In this case, the function automatically transforms the data to the log scale prior to plotting.

The function *range* is a useful auxiliary to plotting. This function takes any number of arguments and returns a vector of length 2, with the lower and upper limit of all the data in the arguments. For example, to plot points at a set of co-ordinates (*x1,y1*) and on the same plot to draw lines through a second set of co-ordinates (*x2,y2*):

```
> plot(x1,y1,xlim=range(x1,x2),ylim=range(y1,y2))
> lines(x2,y2)
```

This makes sure the plot is set up with coordinates large enough to accommodate both (*x1,y1*) and (*x2,y2*).

The *range* function can also be used in conjunction with a square plot in order to force equal size user coordinates in both the x and y directions. For example, if x and y represent distances (as they would if they were output from the *mdscale* function)

```
xy ← mdscale(distmatrix)
r ← range(xy)
par(pty="s") # set up square plot
plot(xy[,1],xy[,2],xlim=r,ylim=r) # force same range on x and y axes
```

The function *abline* draws lines on an existing plot. The lines can be in an intercept-slope form, and in particular can come from a regression data structure that contains a component *coef*.

```
plot(x,y)
abline(reg(x,y)) # add regression line
abline(0,1) # add line with intercept 0, slope 1
```

Optional arguments *h*= and *v*= allow *abline* to draw a set of horizontal or vertical lines; e.g.,

```
abline(h=mean(y))
```

In section 2.4 the commonly used functions *lines* and *points* were introduced. These functions can be used to augment plots produced by the *plot* function. Related functions

```
tspoints
tslines
matpoints
matlines
```

are identical to *tsplot* and *matplot*, except that they only augment a plot with points or lines, and do not change the current user coordinate system.

The function

```
segments(x1,y1,x2,y2)
```

provides the facility to plot sets of line segments. Here the vectors $(x1,y1)$ provide starting points for the line segments, and $(x2,y2)$ provide the end points. A similar function, which plots arrows instead of simple line segments, is *arrows* $(x1,x2,y1,y2)$. Optional arguments to this function control the size and appearance of the arrow. See the detailed documentation.

The functions that are used to add points or lines to a plot (with the exception of *abline*) all recognize whether the previously set up plot had standard or logarithmic axes. If a logarithmic axis was set up, the data are transformed to the log scale prior to plotting, and should be given in the normal, untransformed scale. The functions *rdpen* and *identify* also work correctly in the presence of logarithmic axes.

Most of the time, labelling of a plot can be done with the *title* function, introduced in section 2.4. However, for more precise control of the location of information in the margins of a figure, the function

```
mtext( side, line, label )
```

can be used. The argument *side* specifies whether the text is to be on the bottom (1), left (2), top (3), or right (4) of the plot. The character string in *label* is plotted parallel to the specified side *line* lines of text out from the edge of the plot.

The optional argument *at=* to *mtext* allows text in the margin to be associated with specific user coordinates. For example, if *side* is given as 1 or 3, the argument *at* is interpreted as a set of x-coordinates at which the vector of labels is to be plotted. (The y-coordinates of the text are controlled by the *line* argument). This feature can be used to construct special user-defined axes, etc.

Another option to *mtext* allows the overall labelling of a set of multiple figures. Remember the *outer margins* mentioned in the previous section? The optional argument *outer=* allows the positioning of labelling information in the outer margins instead of in the normal figure margins. For example,

```
par(mfrow=c(3,2),oma=c(0,0,4,0)) # set up array of multiple figures
# with 4 lines of outer margin at top
mtext(side=3,line=0,cex=2,outer=TRUE,
      "This is an Overall Title for the Page")
plot(x,y,main="This is a Title for the Figure")
...
```


Any graphics functions that deal with character strings have the added capability of utilizing embedded *newline* characters that may occur in strings. Whenever this character is encountered, the current plotting position is moved down one line (perpendicular to the string orientation), and the rest of the string is plotted there. In addition, each part of the string between newlines is adjusted (centered, left justified, etc.) independently. The newline character is entered into a string by using the escape sequence "\n":

```
title("This is the first line \nand this is the second")
```

The function *box(n)* draws a box of thickness *n* line widths around the current plot.

In order to build up and add to plots from given data values, it is sometimes convenient to approximate or compute function values at special points. For example, the set of fractions (probability levels) generated in a probability plot of *n* values is returned by *ppoints(n)* or *ppoints(x)* with *x* any vector of length *n*. These values can be used with one of the quantile functions to generate the reference values used in the corresponding probability plot. Suppose, for example, we wanted to do a normal probability plot, but to plot the index of the data value at each point, instead of a plotting character. The *x*- and *y*-coordinates are respectively *qnorm(ppoints(x))* and *sort(x)*. The function *plot* takes *type="n"* to suppress the scatter plot and the function *text* does the desired plot.

```
> Tx ← qnorm(ppoints(x)); Ty ← sort(x)
> plot( Tx, Ty, type="n") #set up axes
> text( Tx, Ty, order(x)) #label with 1:n
```

The function *approx(x,y,xout)* finds approximate values (by interpolation) of the function represented by the data *x* and *y*, at the *x*-values given by *xout*.

3.4.9. Deferred Graphics.

Users with many plots to produce or without access to graphical terminals may find it inconvenient to use the interactive graphics capability in S. For these cases, and where special features of batch graphics devices are needed, S provides a *facility*. This is carried out by a two-part process. First, a deferred graphics file is produced during a normal session with S. This can be done with or without interactive graphics. Once the S session has ended, the file can be plotted on any batch or interactive graphics device.

The production of a deferred graphics file is controlled through the use of a special set of batch graphic device functions or through the function *defer*. The batch device functions are:

Device Type	Features	S Function
Stare-3 plotter		stare
FR80 Microfilm Recorder	high-quality	photo
Prism plotter	color	prism

These functions are used like any other graphic-device function. Any plotting functions executed after the device is specified produce output to the deferred graphics file, but do no interactive plotting.

Interactive plotting and deferred graphics can be combined by means of the function

```
defer(on, ask)
```

The *defer* function should be preceded by an interactive graphics device function, such as those listed in section 2.4. Argument *on* tells whether to turn deferred graphics on (TRUE, the default) or off (FALSE). In addition, if *ask* is TRUE, the user will be asked at the end of each frame whether it should be saved on the deferred graphics file. By default, all output is saved on the file; no questions are asked.

During a deferred graphics session, a symbolic form of the plotting is saved on the deferred graphics file, (which is named *sgraph*.) This file contains lines describing the graphical operations being carried out; details of the format of the file are in the detailed documentation under *defer*. Any further S sessions that utilize deferred graphics will append their output to the file.

To plot the deferred graphics file on a batch device, use the S utility function

S BPLOT device file

Here *device* is the name of the batch graphics device to be used in replotting (the default is *stare*). It is *not* necessary to replot on the same device that was used when the deferred graphics file was created. However, since each device has its own character sizes, etc., the best-looking results will ordinarily be obtained by replotting with the device originally used.

The argument *file* to *BPLOT* is the name of the deferred graphics file to be plotted. By default, the file *sgraph* is plotted, and is also renamed. The renaming prevents further deferred graphics output from being appended to this file. *BPLOT* prints out the new name for the file, so that it is possible to remove it when the plotting job is complete, or to resubmit it on a different device. For example, it may be desirable to get a preview of the plots on the *stare* device, and then submit them for high-quality *photo* output. *BPLOT* also has optional arguments that control system-dependent features such as service grade, etc. See the detailed documentation under *BPLOT*.

It is also possible to replot interactively. For example, suppose that the only graphics device available is a printer, and that the user will have access to an interactive device at some later time. Then,

```
> printer
> defer(ask=T)
Deferred Graphics on - ask about each frame
> plot(height,weight)
> title("Sample plot of Weight vs Height")
> show
... printer plot produced here ...
> qqnorm(weight)
Save last frame? yes           refers to height,weight plot above
> ...
```

This session produces the deferred graphics file. Later, on an interactive device, e.g., a Tektronix 4014 scope,

```
> tek14
> replot                      # re-plots all frames from save file
```


The *replot* function turns off deferred graphics to prevent appending to the same file it is reading. Deferred graphics mode is also turned off whenever a new device is specified or by the expression *defer(FALSE)*. In both cases, the message "Deferred Graphics turned off" will be printed as a warning.

The S Macro Processor

4. The S Macro Processor.

The S macro facility provides a means for extending the S language. It allows new function-like operations to be built up from existing S expressions, and also allows commonly used expressions to be saved and executed with a minimum of typing.

Section 4.1 provides an introduction to S macros. Basic techniques for defining macros are covered in section 4.2. Sections 4.3 through 4.7 discuss more advanced topics in macro design. Finally, section 4.8 shows a large collection of macros, including many of the macros available on the shared database.

4.1. Macros in S.

Macro calls in S consist of the macro name, preceded by the character "?" to mark the call as a macro, and followed optionally by arguments to the macro, in parentheses. For example,

```
> colmean — ?col(x,mean)
```

The text of the expression in which the macro call appears is passed to the S macro processor. Finding "?col" in the expression causes the macro processor to search for a macro named *col*. The definition of a macro in S is stored on an S database; the definition of macro *col* is expected to be in a dataset named *mac.col*. This particular macro is available to all users, since *mac.col* is on the shared database. Once the definition of *col* has been found, the macro processor scans the arguments in the call to the macro.

The text for the arguments is substituted into the text of the macro, and the output resulting from the substitution is rescanned, to look for further macro calls. If the scan proceeds without finding any more macro calls, the resulting text is written to a file and the file is parsed and executed by S through a hidden use of the *source* function.

Keep in mind that the whole process of expanding macro calls and of substituting arguments into the macro text takes place without evaluating any S expressions. In particular, macros operate only on the text of their arguments, without knowing the contents of datasets or anything else about the effect of the S expressions.

Some distinctive features of the approach to macros in S are relevant. Macro definitions in S are kept individually in S databases and are read in only when the specific macro is invoked. There is essentially no penalty (other than disk space) for having many macro definitions of which only a few are likely to be used in any expression. Although macros on databases can be treated like any other S data structures, there are special S functions *define*, *mprint* and *medit* to generate, print and edit macro definitions. Macros are stored under the special prefix "mac.". While legal, it is probably unwise to use names beginning "mac." for anything other than macro definitions. The argument to *medit* or *mprint* should be the dataset name: to see the definition of macro *col*, type *mprint(mac.col)*.

Macro calls in S are similar to but not identical to S function calls. Like functions, macros are called with arguments given either positionally or by name. One obvious difference is the requirement that macro names be preceded by "?". Although this requires typing an extra letter, there are several advantages. It is possible, and

often convenient, to have a macro with the same name as an S function; for example, we will show below a macro *reg* which is used to do a specialized form of regression analysis, including a call to the S function *reg*. Having "?" preceding the macro call ensures that this usage is unambiguous.

The use of "?" also saves execution time by not passing to the macro processor any expressions which do not include macro calls. The "?" also may help to clarify in the user's mind which options and arguments are being processed at *macro time* and which at *function execution time*. Resolving options at macro time may contribute significantly to overall efficiency. We will see some examples below.

4.2. Defining Macros.

Here is the definition of a simple macro, *reg*, which does a regression and then produces two diagnostic plots (a normal q-q plot of residuals and a plot of the absolute values of residuals against fitted values):

```
MACRO reg(x,y)
$T ← reg(x,y)$resid
qqnorm($T)
plot(y-$T,abs($T)) #abs resid vs fitted
END
```

The definition consists of a MACRO statement giving the name of the macro and its arguments, then the S expressions forming the macro text, and finally the END statement. The text generally involves the arguments to the macro.

A macro can be created initially by typing (with a text editor) the definition as shown into a file, say "regmac", and then using the S function *define*:

```
> define("regmac")
```

The macro definitions on the file will be processed and saved; in this case as a dataset *mac.reg* on the user's save database (an optional argument, *pos*, to *define* can be used to put the definition on another database; e.g., *pos=1* for the working database). Notice that the name of the macro determines the name of the dataset. The name of the file from which the definition is read has no effect on the macro name. Any number of macro definitions can be read from a single call to *define*. As they are read from the file, the *dataset name* for each macro is printed. If no file argument is given to *define*, the macro definitions will be read from the user's terminal. In this case, the user will be prompted by "N>" for the line containing the MACRO name and with "D>" for lines containing macro definition. When the user hits carriage return in response to "N>", *define* finishes. When the macro(s) to be defined are extensive or complicated, it is better practice to prepare them with a text editor, on a file, and use *define* only when you are satisfied with the appearance of the macros.

The macro *reg* is an example of a useful, simple macro. The advantages of the macro to the user are that all three expressions can be invoked at once, saving most of the typing and possible errors. For now only the *resid* component produced by the S function *reg* is assigned, since the other portions of the returned value are not used later.

The use of *\$T* as the name for the intermediate dataset is a suggested stylistic touch. We shall follow the convention of beginning all intermediate dataset names in macro definitions with "\$T". Explicitly beginning the name with "\$" ensures that, even if the user has a prefix in effect (section 3.2), all of the intermediate datasets will be

placed under the prefix "T". Therefore, users should not give important datasets names like "Tx", "Ty", etc. We will show later on how these intermediate datasets can be removed at the end of the macro.

Once defined, the text of a macro can be printed by the function *mprint*:

```
> mprint(mac.reg)

MACRO reg(x,y)
$T ← reg($1,$2)$resid
qqnorm($T)
plot($2-$T,abs($T)) #abs resid vs fitted
END
```

The only substantial difference from the original is that argument names in the text have been changed to the positional form: *\$1* for the first argument and *\$2* for the second argument. This is accomplished in the *define* function, which scans the macro definition for all occurrences of the argument names, and replaces these with the positional symbols. On the other hand, it is perfectly legal to use the *\$1*, *\$2* form in the macro definition.

It is also possible to give a list of the names of macros to be printed to *mprint*. For example

```
mprint(list=list("mac.*",pos=2))
```

prints all the macros on the save database. An optional argument *file=* to *mprint* allows macro definitions to be written to a file.

To edit the text of a macro, use the S function *medit*:

```
> medit(mac.reg)
```

medit invokes the text editor *ed*, with the text of the macro as the file to be edited. Writing an edited version of the file (without specifying a file name) and quitting saves the new version. Since the new text is reprocessed through *define*, the edited changes can refer to the arguments by name. To create a *new* macro with the edited text (and leave the original available) just change the name of the macro on the MACRO statement.

Let us edit the macro *reg* to illustrate some other possibilities with macros. We can make the second plot optional by introducing an argument *plotfit* as the third macro argument, and changing the last expression to

```
if(plotfit)plot(y-$T,abs($T))
```

If we want the plot to be done by default, *plotfit* can have a default value of TRUE. Default values for macro arguments are strings supplied after the argument name in the MACRO statement, enclosed in slashes:

```
MACRO reg(x, y, plotfit/TRUE/ )
```

The default text can contain other macro calls. It cannot, however, contain references to the other arguments; a use of *x* or *y* would be left untranslated (see section 4.3).

So far, our *reg* macro has thrown away the regression data structure (except for residuals). A better approach is to return the regression structure as the value of the macro, so that the macro then acts as an extended form of the corresponding S function. The technique to use here is the *compound expression* (see section 3.1.3). If the

text of the macro is made into a compound expression, the value of the last sub-expression becomes the value of the compound expression and, therefore, the value of the macro. To be even more sure that the macro can be used as a function, the compound expression in turn should be enclosed in parentheses. Among other effects, this ensures that the macro text will not be broken up by its context. For example, a macro *abc* with text " $\$1+\2 " would not produce the intended result in the expression

```
?abc(x,y)*2
```

since the expanded text would be

```
x+y*2
```

The general rule in defining a macro which should act like an S function is to begin the text with "({", end with "})" and return the intended value as the last sub-expression.

If an argument appears more than once in the text of a macro, it is generally desirable to evaluate it once and save the value in an intermediate dataset. Not only may this save considerable execution time, but some uses of arguments might lead to illegal S expressions if the argument supplied by a user was an arbitrary expression.

In our regression example, we can assign the regression to $\$T$ and return this as the value. Here is the macro, with the changes we discussed edited in.

```
MACRO reg(x, y, plotfit/TRUE/ )
({ $Ty ← y
  $T ← reg(x,$Ty)
  qqnorm($T$resid)
  if(plotfit)plot($Ty-$T$resid, abs($T$resid))
  $T
})
END
```

Because y was used twice in the text of the macro, we have created an intermediate dataset $\$Ty$ for its value.

We can remove the intermediate datasets $\$T$ and $\$Ty$ at the same time we return $\$T$ as the value of the macro. The S function *rm* will remove its arguments from the working database. By using the special argument *value*, *rm* can also return any desired expression. The last line of the macro definition is then

```
rm($T,$Ty,value=$T)
```

One more change will illustrate some of the flexibility of the use of macro arguments. Suppose we want our plots to have an informative title; for example, the call to *qqnorm* could be changed to:

```
qqnorm($T$resid,sub="Normal q-q plot",
main="Regression of y on x")
```

The point to note is that x and y are substituted from the actual macro arguments into the character string used as the main title. The call to the macro of the form

```
?reg(pred[,1:4],log(response))
```

would produce as its main title on the plot:

"Regression of log(response) on pred[,1:4]"

In this way, macros used to generate plots and printing can be made considerably more informative than simple function calls. The macro is now:

```
MACRO reg(x, y, plotfit/TRUE/ )
({ $Ty ← y
$T ← reg(x,$Ty)
qqnorm($T$resid,sub="Normal q-q plot",
main="Regression of y on x")
if(plotfit)plot($Ty-$T$resid, abs($T$resid))
rm($T,$Ty,value=$T)
})
END
```

The discussion of this and the previous section should be sufficient to let you use the macros on the shared database, or to define and use some macros of your own. You may wish to do this before reading any further. Also, a number of examples of macros, with explanatory comments, are given in section 4.8. Documentation for shared macros is given on-line by the S function *help*; e.g.,

```
help(macro="col")
```

for the documentation of macro *col*. Section 3.2.7 describes utilities that can be used to construct documentation of your own macros.

4.3. Literals; Conditional Expansion.

Sometimes it is necessary to suppress the interpretation of some text in the macro definition. The most common case is that the name of one of the macro arguments must be used literally; e.g., it is the same as the name of an argument to an S function. In the *reg* macro, if we had named the third argument *plot* rather than *plotfit*, there would be trouble calling the S function *plot*, since *define* would turn all occurrences of "plot" into "\$3". A built-in macro construction is used to avoid this:

```
?(anything)
```

passes *anything* on without scanning it. Literals are recognized both when the macro definition is processed by the *define* function and when the macro text is processed to substitute the arguments. Literals can protect characters in the default text for macro arguments (principally, this allows the character "/" to appear in the default string), and they can protect argument names or other special characters, such as "?" or ",", within the macro text. In our example,

```
.if(plot)?(plot)( ... )
```

would solve the problem. This points out another useful feature: macro expressions need to be legal S expressions only *after* expansion. The strange use of parentheses above does no harm.

The use of literals in the example protected a name during the definition. Using a literal to protect a special character during expansion is a little more subtle. The contents of the literal are passed on to the macro output without processing, but with "?"(" and ") stripped off. This use of literals is less frequent and will be deferred to the examples in section 4.8.

The *if* statement in the *reg* macro is evaluated at function execution time. This has the advantage that the argument *plotfit* may be an arbitrary expression, but the disadvantage that more interpretation of the macro output is needed when the resulting expression is evaluated. An alternative is to use a conditional macro *IFELSE*. In its simplest form, this has four arguments. If the first two are equal (as strings at macro time), it expands into the third argument; otherwise, it expands into the fourth argument. In our example, suppose the plotting is to be done only if the *plotfit* argument is TRUE:

```
?IFELSE(plotfit,TRUE, plot($Ty-$T$resid, abs($T$resid)) )
```

Here, the fourth argument to *IFELSE* is implicitly empty. When *?IFELSE* is used, equality is determined at macro time, so that the test cannot depend on the value of an S expression. It is a character-by-character test for equality: "T" would not be equal to "TRUE", even though at execution time they both stand for the logical value true. If we replace the *if* statement in the example with the *?IFELSE* form, the user of the macro could not type

```
?reg(x, y, rss<.8)
```

and expect the macro to expand reasonably.

A more powerful example of conditional expansion solves a problem mentioned in the previous section: how to make default values depend on other arguments. Suppose we want the default test for *plotfit* to be *len(y)>50*. This would not work as the default text in the MACRO statement because default text is not scanned on substitution. However, we can generate the same result by not supplying a default string, and then testing for a null argument:

```
MACRO reg(x, y, plotfit)
$Tpl← ?IFELSE(plotfit,,len(y)>50,plotfit)
if($Tpl) plot(...)
```

Now *\$Tpl* represents either the macro argument or the default test. This time, we want the default test to depend on data values; therefore, the test must be done at execution time by *if*, not at macro time by *?IFELSE*.

There is a more general form of *?IFELSE* which takes an arbitrary number of arguments, in sets of 3. If the first two of each set match as strings at macro time, the expansion of *IFELSE* is the third argument in the set. Otherwise the first two of the next set are compared, and so on. If one final argument is given, this is the default expansion if all matches fail. For example, to allow "YES" and "NO" as values of *plotfit*:

```
$Tpl ← ?IFELSE(plotfit,YES,TRUE,
plotfit,NO,FALSE,
plotfit,,len(y)>50,
plotfit)
```

Notice that leading white space in arguments to macros is ignored, so that the macro calls can have a natural layout. Trailing white space is retained; see the example at the end of section 4.4.

4.4. Macros with Many Arguments.

As with S functions, some macros will want to have arbitrarily many arguments. This is managed by including in the MACRO statement only as many arguments as should be recognized specially, and then matching all the remaining arguments by the special construction "\$*". For example, here is a form of the *reg* macro which takes one y variable and an arbitrary number of x variables:

```
MACRO reg( y )
({ $Tx ← cbind($*)
  reg($Tx,y)
})
END
```

(For the moment, we have omitted the plotting.) The occurrences of "\$*" in the text of the macro are replaced by all the actual arguments to the macro (if any) which have not been matched to any of the arguments in the MACRO line. These remaining arguments replace the "\$*", separated by commas. In this case, for example, calling the macro in the form *?reg(resp,var1,var2,x)* expands the first line of the definition into

```
({ $Tx←cbind(var1,var2,x)
```

This form of construction is useful when one wants to call S functions with arbitrarily many arguments. Also, a macro calling a graphics function may wish to include graphics parameters (section 3.4.6) in the macro call. The macro *symplot* below generates a plot of the sorted values of a vector against the same values in the reverse order (a plot which can be used to test for symmetry):

```
MACRO symplot(x)
$Tx ← sort(x)
plot($Tx,rev($Tx),xlab="x",ylab="rev(x)",$*)
  #plot with graphics parameters
END
```

The macro call *?symplot(resid,pch="+",col=3)* passes the graphics parameters *pch* and *col* to the *plot* function.

One problem with having arbitrarily many arguments is how to allow special arguments with default values. How does one reintroduce the argument *plotfit* into the *reg* macro, without having it match the first of the x variables? In both S functions and macros, the usual solution is to require the argument to be supplied in the form *name=value*. To restore optional plotting to the regression macro we reintroduce the argument with a trailing "=":

```
MACRO reg( y, plotfit=/TRUE/)
```

with the same sort of tests as discussed in the previous section to determine whether to do the plotting. With the macro defined in this way, the user must use

```
plotfit=FALSE
```

in the macro call to suppress the plotting.

The expansion of "\$*" can be generalized to use the list of arguments in any way at all, through the built-in macro *LOOP*. This macro concatenates instances of a string (its first argument) in which the remaining arguments are substituted successively. It is effectively a macro-within-a-macro, in that the string is a (limited) form of macro

definition and the remaining arguments provide instances of this definition. Here is a simple application: suppose we want to create a vector which has all the macro arguments combined as quoted strings. The purpose is to create a vector of character names in S from the list of arguments to the macro. The *LOOP* macro does this if its first argument contains "%1". Each of the other arguments to *LOOP* is then substituted for "%1" in the first argument, and the results concatenated together, with commas in between:

```
?LOOP('%1',x+y,-3,log(resp))
```

for example, expands into

```
'x+y','-3','log(resp)'
```

The desired trick of making the vector of names is then accomplished by the macro *names*, defined as follows

```
MACRO names
c(?LOOP("%1", $*))
END
```

More complex applications of *LOOP* are also possible. Sometimes it is desired to generate separate expressions for each of the arguments. A macro to produce separate scatter plots of all other arguments against the first argument would be:

```
MACRO mplot(x)
$T ← x
?LOOP( plot($T,%1,xlab="x",ylab="%1")
,sep=,$*)
END
```

In this case, the first argument to *mplot* is *x*. Each of the remaining arguments to *mplot* is substituted for both instances of "%1" in the *plot* call. Notice that the first argument to *LOOP* is "plot(.....)" and is terminated with a new-line. The special argument *sep=* to *LOOP* gives a separator different from the default comma. Here the separator can be null because we put a new-line at the end of the first argument to *LOOP* (the preferred way to separate statements generated by *LOOP*). The output is more readable on separate lines, and the danger of overflowing text buffers during parsing is reduced. Thus

```
?mplot(pred[,1],resp,log(resp),sqrt(resp))
```

expands into

```
$T ← pred[,1]
plot($T,resp,xlab="pred[,1]",ylab="resp")
plot($T,log(resp),xlab="pred[,1]",ylab="log(resp)")
plot($T,sqrt(resp),xlab="pred[,1]",ylab="sqrt(resp)")
```

In this example again, we have taken advantage of the macro processor to generate informative labels, and have copied the argument *x* into *\$T* for efficiency, since it is used repeatedly. Since this macro returns no useful result, the enclosing "({" and "})" are omitted.

4.5. User-Oriented Macros.

Macros that are easy for others to use may need to communicate with the user during macro expansion. The built-in macros *MESSAGE*, *FATAL* and *PROMPT* all take a message to be printed on the user's terminal when the macro is expanded. In addition, *FATAL* induces an error which terminates the evaluation of the expression containing the macro call. *PROMPT* produces, as its expansion, a line of input read from the user's terminal in reply to the prompt. *MESSAGE* produces no expansion text.

The *MESSAGE* macro can be used just to print a message to the user:

```
MACRO learn
?MESSAGE(Hello, this is the learn macro)
...
```

More typically, all three macros will be used conditionally to report special situations or establish values for arguments. The *FATAL* macro is one way to make arguments obligatory, or to check other problems:

```
MACRO reg( x/?FATAL(Must supply argument x)/, ...
```

The result is that, at macro expansion time, the user who omits the argument *x* will see the error message. As a general approach, this is better than just allowing expansion without an essential argument. The best that can happen in the latter case is an S syntax error. Not only does this take longer, but the error is usually obscure to the user, particularly if it comes far into the expansion of a complicated macro. Even worse, the S expression may turn out to be syntactically correct but produce ridiculous results.

While *FATAL* is better than nothing, one may give the macro user a more friendly response in most cases by the use of *PROMPT*. The message given to *PROMPT* is printed on the terminal. The user replies with a line of input, which, not including the new line, becomes the expanded form of *PROMPT*.

```
MACRO reg(x/?PROMPT(X matrix for regression:)/,
...
```

Thus, a user who had totally forgotten how to use this version of the macro *reg* might go through:

```
> ?reg
X matrix for regression: pred[,1:5]
Y vector for regression: log(resp)
Plot fitted vs abs. resid (T or F)? T
...
```

It is a question of style or psychology as to how far such prompting is desirable. So long as users can get out of the prompting session (by hitting interrupt), the use of prompts seems a convenience.

In order for more complicated macros to be usable, there is a need for documentation. See for example, the macro detailed documentation for system macros in section 9. Section 3.2.7 describes how to document both datasets and macros.

Macros may be made self-documenting through the use of the *SYS* built-in macro which executes an operating system command. For example


```
?IFELSE(?PROMPT(Do you want Instructions?),yes,?(?SYS(cat instruct)))
```

will list the file "instruct" if the user requests instructions.

The style advocated earlier, of assigning all arguments to temporary datasets if they appear more than once in the macro text, becomes obligatory if *PROMPT* is used to supply default values. Otherwise, the user will be prompted with the message *every* time the argument appears (and the macro processor will cheerfully substitute different responses each time!). Occasionally, it may be impossible to assign arguments, as when the argument is part of a function or dataset name. In this case, the use of *PROMPT* requires that the macro come in two layers. The outer layer contains the arguments and the *PROMPT* defaults, but its text consists only of the invocation of the inner layer:

```
MACRO qqplot( x,dist/?PROMPT(What distribution?)/)
# create quantile-quantile plot against arbitrary distribution
?qqplot2(x,dist)
END
```

```
MACRO qqplot2(x,dist)
$Tx ← x;
plot(?(q)dist(ppoints($Tx)), sort($Tx),
      xlab="Ordered quantiles of dist distribution",
      ylab="Sorted values of x")
rm($Tx)
END
```

The prompt will occur only in preparing the macro call to *qqplot2* so that each message will be issued no more than once. See section 4.6 for another means of accomplishing this end.

Readers who have come this far in the macro discussions may find it useful to understand more details of how default substitution works. Suppose the macro processor receives the line

```
?qqplot(abc)
```

It begins by reading the dataset containing the definition of *qqplot*, and extracting the text, argument names and default argument text as processed by *define*. If the macro name is followed by a parenthesized argument list, the arguments in this list are now scanned. If another macro call is encountered during the scan of the arguments, this definition is read in and the process of argument scanning now proceeds on this lower level. Eventually, for some macro, the actual arguments will be pure text. These are matched to the argument names, and the default text is used for any missing arguments. But (this is critical here) the default text is treated as part of the macro definition, and is not scanned at this stage.

The argument text is substituted into the macro definition text, and the result is pushed onto an internal buffer. It is not output yet, because the text of *qqplot* is free to contain other macro calls. If the current macro was part of an argument to another macro, the process of expansion continues. When the top level expansion is complete, the macro processor rescans the pushed-back text, and at this point will pick up macro calls in the definition. In particular, the macro calls to *PROMPT* are now part of the actual arguments to *qqplot2*. Therefore, they will be evaluated (once) before the text of *qqplot2* is expanded.

The internal buffers used by the macro processor are limited in size, since most macros do not require large amounts of space. However, if you have particularly large macros, it may be necessary to enlarge the space given to these internal buffers. The expression

```
options(macsize=4000)
```

would double the size of the macro buffer from its default size of 2000.

This detailed knowledge of how the macro processor works is not needed for most applications. Occasionally, it becomes necessary to understand such techniques as delaying macro expansions or protecting strings with "?(...)". For further reading, see the macro processor discussed in *Software Tools* (Kernighan and Plauger, 1976), on which the macro expansion part of the S macro processor is based.

4.6. Temporary Definitions; Recursive Calls.

Macro definitions generated through the S function *define* are permanent definitions, created outside the macro processor and saved on an S database. It is possible, for specialized requirements, to define macros during the macro expansion itself. This is accomplished by the *DEFINE* macro:

```
?DEFINE(name,text)
```

creates a macro with *text* as the definition. (It is very difficult to create a macro with arguments in this fashion.) Throughout the current invocation of the macro processor, the macro may be invoked as *?name*. The macro is not saved permanently, however. Once the macro processor has expanded the expression given to it, all temporary macro definitions disappear.

Temporary definitions are useful mainly as a mechanism to save some information during the expansion of a macro, which will then be re-used later on in the expansion. For example, the use of *PROMPT* shown with a two-layered macro in the previous section could also be handled by defining the value of the prompt as a temporary macro. The technique will be to prompt the user of the macro, if the argument *plots* is omitted, to specify whether a q-q plot, a plot of residuals, both or neither is wanted. The first character of the response is extracted, by a built-in macro *SUBSTR* (section 4.7), and defined temporarily as the macro *plotopt*. In the remainder of the definition of *reg*, the temporary macro can be invoked every time the user's response is needed. The extended form of the *IFELSE* macro can then be used to generate one or the other or both of the plotting expressions.


```

MACRO reg(x,y,
  plots/?PROMPT(Plotting (qq, residual,both or none):)/
)

({ $Ty ← y
$T ← reg(x,$Ty)
?DEFINE(plotopt,?SUBSTR(plots,1,1))
?IFELSE(
  ?plotopt,q,qqnorm( $T$resid ),
  ?plotopt,r,plot($Ty-$T$resid, abs($T$resid), xlab="Fitted"),
  ?plotopt,b,qqnorm( ... )
  plot( ... )
)
rm($T,$Ty,value=$T) })
END

```

4.7. Built-In Macros and Special Constructions.

Built-in macros are part of the S macro processor itself: they are not created on a database as a result of *define*. They have appeared in previous sections, but the definitions are collected here, in a general form.

IFDEF(name,s1,s2)

If macro *name* is currently defined, this macro expands as *s1*, otherwise it expands as *s2* (or null if *s2* is .

IFELSE(a1,b1,c1, a2,b2,c2, ..., d)

The macro takes $3*k$ or $3*k+1$ arguments. If the arguments *a1* and *b1* match as strings, the macro expands as *c1*. Otherwise, arguments *a2* and *b2*, if both present, are compared, and so on. If all matches fail, the macro expands as *d*, which is implicitly null if $3*k$ arguments were given.

LOOP(string, a1, a2, ...)

The output of the macro consists of the contents of *string* repeated as many times as the number of remaining arguments *a1*, etc. allow. Successive repetitions are separated by a separating character (comma by default) which may be given by the *sep=* argument. In order to separate by new lines, put a new line at the end of *string* and specify the separator as null.

The contents of *string* are scanned for the special tokens "%1", "%2", etc. Let *m* be the largest integer which appears in such a token. Then the first *m* arguments from *a1*, etc. are substituted into the first occurrence of *string* to replace "%1", etc. The next *m* arguments are substituted into the next occurrence, and so on until the list of arguments has been used. (Obviously, values of *m* are usually just 1 or 2. The argument *skip=* can also be used to specify *m*.

DEFINE(name,text)

When an invocation of this macro is encountered in the course of scanning the text of another macro, *name* is entered into the table of currently known macros, with a definition consisting of *text*. The contents of *text* are not scanned for argument names or defaults. All arguments in *text* must be in the form "\$1", etc., or "\$*", and all missing arguments have null default text. The macro so defined will normally be invoked later in the text of the macro containing the invocation of *DEFINE*. Other, more subtle uses of the temporary definition mechanism are

possible but discouraged because of the unpredictable effects produced.

SUBSTR(string,start,length)

Returns the substring of its first argument, starting at character position *start* and continuing for *length* characters (character positions are indexed from 1). If the last argument is omitted, the rest of the string is returned.

SYS(string)

Executes *string* as a command to the operating system. Useful for printing a file of instructions, etc.

\$*

Notation for "all arguments to this macro that were not matched by formal parameters". These arguments are substituted for "\$*", and are separated by commas.

?(string)

? (

Quoting convention for the macro processor. Each time a string enclosed in "?(...)" is encountered, the "?(" and ")" are stripped off, and *string* is not evaluated.

4.8. Examples of S Macros.

The macros listed in this section are available on the system shared database.

allows easy editing of macros, even with prefix in effect

```
MACRO medit(name)
```

```
medit(?( $mac.)name)
```

```
END
```

Lists names of all macros on database

```
MACRO mlist(pos/2/)
```

```
ls("$mac.*",?(pos)=pos)
```

```
END
```

Constructs documentation file for macro or dataset

```
MACRO prompt(name,pos=/2/)
```

```
if(c(ls("$mac.$1",?(pos)=pos),"x")=="x"){
```

```
    print("Creating Dataset Documentation for $1 on file $1.d")
```

```
    dprompt("$1",ls("$1*",?(pos)=pos))
```

```
} else {
```

```
    print("Creating Macro Documentation for $1 on file $1.d")
```

```
    mprompt(get("mac.$1",?(pos)=pos))
```

```
}
```

```
END
```

union of several sets

```
MACRO union
```

```
uniq(c($*))
```

```
END
```


intersection of two sets

```
MACRO intersect(x,y)
({$Tx ← uniq(x)
rm(list="$Tx",value=$Tx[!NA(match($Tx,uniq(y)))])
})
END
```

produces list of indices where condition is true

e.g. ?which(x>3)

```
MACRO which(condition)
({ $T ← condition;
rm(list="$T",value=(seq($T))[$T])
})
END
```

produces a sample from x with a probability of alpha

of selecting any given value in x

```
MACRO probsamp(x, alpha/?PROMPT(Sampling probability:)/)
({$Tx ← x
rm(list="$Tx",value=$Tx[runif(len($Tx))>alpha] })
END
```

Generates a random permutation of x

```
MACRO rperm(x)
({$Tx ← x
rm(list="$Tx",value=$Tx[order(runif(len($Tx)))]) })
END
```

Generates a random sample of size n from x

```
MACRO sample(x,n/?PROMPT(Sample size:)/)
(?rperm(x)[seq(n)])
END
```

add a comment to a data set

```
MACRO comment(x)
if(ncomp(x)==1) x ← cstr(comment="$*",Data=x$[1]) else
x ← mstr(x,comment="$*")
END
```

applies function fun to the columns of x

fun must be unquoted name of the function, cannot be variable

Example: ?col(mymatrix,mean)

```
MACRO col(x,fun/?FATAL(must supply function name)/)
apply(x,2,"fun",)$*
```

END

Applies function fun to the rows of matrix x
Handles the transposition problem of row apply
if fun returns vectors

MACRO row(x,fun/?FATAL(must supply function name)/)

{ \$Tx ← apply(x,1,"fun",\$*)

if(ncol(\$Tx)>1) \$Tx ← t(\$Tx) # transpose if necessary to make same shape as x

rm(list="\$Tx",value=\$Tx)

}

END

Sweeps summary statistics out of the rows of matrix x

Example: ?rsweep(data, ?row(data,mean))

removes row means from matrix data

MACRO rsweep(x,summary,fun/-/)

sweep(x,1,t(summary),"fun",\$*)

END

Sweeps summary statistics out of the columns of matrix x

MACRO csweep(x,summary,fun/-/)

sweep(x,2,summary,"fun",\$*)

END

Sweeps the summary statistics x out of the structure z

Example: Given structure s with a number of vector components

?ssweep(s, sapply(s,"mean"))

removes means from each component

MACRO ssweep(z,x,op/-/)

{ \$Tz ← z # sweep summary x out of structure z

\$Tgroup ← rep(1:ncomp(\$Tz), sapply(\$Tz,"len"))

mapping of component memberships

\$Tout ← c(\$Tz) op x[\$Tgroup]

rm(list=c("\$Tz","\$Tgroup","\$Tout"),
 value=split(\$Tout, \$Tgroup))

}

END

Applies a macro to all other arguments of this macro

Example: if the macro analyze does an analysis of a single set of data

and we want to analyze the data sets x,y,z

?apply(analyze,x,y,z)

is equivalent to

?analyze(x)

?analyze(y)


```

?analyze(z)
MACRO apply(macro)
?LOOP(?($1($%1)), $*, sep=
)
END

```

All subset regressions using the Cp statistic
Produces Cp plot and then allows user to identify points
on the plot, for which it then provides full regression analyses

```

MACRO Cp(x,y,wt,int,names,plot/T/,identify/T/)
$Tx ← x; $Ty ← y    # make sure args evaluated only once
$Tleap ← leaps($Tx,$Ty,?(wt)=wt,?(int)=int,?(names)=names)
if(!plot)source()    # exit from macro
plot($Tleap$size,$Tleap$Cp,type="n",ylab="Cp Value",
     xlab="Number of Variables",
     xlim=range(0,$Tleap$size+1),log=?( "y"))
text($Tleap$size,$Tleap$Cp,$Tleap$label)
lines(seq(max($Tleap$size)+1))
if(!identify)source() # exit from macro
"identify the regressions you want to see"
for(j in identify($Tleap$size,$Tleap$Cp,plot=F))
  regress($Tx[$Tleap$which[j]], $Ty,?(wt)=wt,?(int)=int,?(names)=names)
rm(list=c("$Tx", "$Ty", "$Tleap"))
END

```

Adjusted variable computation
?adjust(x,y,which) adjusts x[,which] and y for all
other x variables

```

MACRO adjust(x,y,which/?PROMPT(adjust which x variable?)/)
({ $Tx<-x; $Ti<-which; $Txi<-$Tx[,-$Ti]
rm( value=cstr(
  ?(x)=reg($Txi,$Tx[$Ti],$*)$resid,
  ?(y)=reg($Txi,y,$*)$resid),
  list=c("$Ti", "$Tx", "$Txi")
)
})
END

```

two sample rank test statistic

```

MACRO ranktest(x,y)
sum(rank(c(x,y))[seq(x)])
END

```

For use in regression - finds where NAs are in observations

Use: $m \leftarrow ?missing(x,y)$
 $regress(x[!m,],y[!m])$ # omit missing data

```
MACRO missing(x,y)
?row(NA(cbind(x,y)),any)
END
```

*creates individual vectors from columns of matrix
called with matrix and vector of column names*

```
MACRO mat2vecs(x,names)
$Tx ← x; $Tnames ← names
for(i in seq($Tnames))
    assign($Tnames[i],$Tx[,i])
rm($Tx,$Tnames)
END
```

*creates matrix out of list of vectors
also creates list of column labels out of vector names*

```
MACRO vecs2mat(name)
name?(.data) ← cbind($*)
name?(.collab) ← c(
    ?LOOP("%1", $*)
)
END
```

creates distance structure from full matrix

```
MACRO full2dist(x)
({$Tx ← x
rm(list="$Tx",
    value=cstr(Size=nrow($Tx),
        Data=$Tx[row($Tx)>col($Tx)])
})
})
END
```

creates full matrix from distance structure

```
MACRO dist2full(x)
({$Tx ← x
$TSize ← $Tx$Size
$Tfull ← matrix(0,$TSize,$TSize) # matrix of zeroes
$Tfull[row($Tfull)>col($Tfull)] ← $Tx
rm(list=c("$Tx", "$Tfull", "$TSize"),value=$Tfull+t($Tfull))
})
END
```

produce scatter plot with id numbers at each point

produce reasonable labels if x and y are expressions

and allow points to be marked in character size different

from standard

```
MACRO idplot(x,y,xlab=,ylab=,cex=1/)
$Tx ← x
$Ty ← y
plot($Tx,$Ty,$*,
     ?(xlab)=?IFELSE(xlab,,"x",xlab),
     ?(ylab)=?IFELSE(ylab,,"y",ylab),
     type="n")
text($Tx,$Ty,seq($Tx),?(cex)=cex)
rm($Tx,$Ty)
END
```

runs extract utility and restores from .ext file to database

```
MACRO extract(name)
!S extract name
restore("$1.ext",$*) # allows pos=, print=
END
```

*Discriminant analysis of x matrix where each row of x
is in the group specified by vector group*

```
MACRO discr(x,group)
({
$Tg ← group
$Tsize ← table(code(sort($Tg)))
rm($Tsize,$Tg,
   value=discr(x[order($Tg),], $Tsize)
)
})
END
```

plot macro for result of multi-dimensional scaling

Preserves distances in plotted output, and identifies each object

```
MACRO mdsplot(mds)
$Tmds ← mds
$Tx ← $Tmds[,1]; $Ty ← $Tmds[,2]
$Tr ← range($Tx,$Ty)
par(pty="s") # would be better to query, restore at end
plot($Tx,$Ty,xlim=$Tr,ylim=$Tr,type="n",
     xlab="Dimension 1",
     ylab="Dimension 2"
)
text($Tx,$Ty,seq($Tx))
END
```

bar plot of category

```
MACRO bartable(category)
$T ← table(category)
barplot($T$Data,names=$T$Label$[1],
        xlab=compname($T$Label))
rm($T)
END
```

Produces all pair-wise scatter plots of columns of matrix x

```
MACRO pairs(x,labels,head)
$T ← x; $Tp ← ncol($T)
$Tl ← ?IFELSE(labels,,encode("Var.",1:$Tp),labels)
$Thead ← ?IFELSE(head,,"Pair-wise Plots of x",head)
par(oma=c(0,0,3,0))
mtext($Thead,3,1.5,outer=T)
par(mfrow=rep($Tp-1,2),$*)

for(j in 2:$Tp) {
  for(i in seq($Tp-1)) {
    if(j>i)plot($T[,i],$T[,j],xlab=$Tl[i],ylab=$Tl[j]) else frame()
  }
  frame()
}
par(mfrow=rep(1,2))
rm($T,$Tp,$Tl,$Thead)
END
```

create vector of character strings from arguments

```
MACRO names
c(?LOOP("%1",*))
END
```

create quantile-quantile plot against arbitrary distribution

prompts user if distribution is not given

```
MACRO qqplot(x,dist/?PROMPT(What distribution?)/)
?qqplot2(x,dist,$*)
END
```

```
MACRO qqplot2(x,dist)
```

```
$Tx ← x
plot(? (q)dist(ppoints($Tx),$*),sort($Tx),
     xlab="Ordered quantiles of dist $* distribution",
     ylab="Sorted values of x")
rm($Tx)
END
```


produces list of database names and contents

MACRO listall

```
$T ← search()      # all databases
for(i in 1:len($T)){
    print(encode("Database",i,$T[i]))
    print(list(pos=i,$*))
}
rm($T)
END
```

produces list of the names of all functions

UNIX only -

MACRO listfun

```
{
sys(encode("ls ",search(1),"/x >>/tmp/allfun",sep=""))
$T_read("/tmp/allfun",mode=CHAR,print=F)
sys("rm /tmp/allfun")
rm($T,value=$T)}
END
```

..... The following macros work together as a group

The user interface is the macro cleanup which takes a pattern

that gives the names of datasets that might be candidates for elimination

It uses the define macro to create a macro with the names of all data

sets that might be removed.

Then the macro rmlist is used to remove an entire list of data sets,

asking permission to remove each one.

MACRO prmd(x)

```
?DEFINE(answer,?SUBSTR(?PROMPT(OK to remove x),1,1))
?IFELSE(?answer,Y,rm(x,print=T),
    ?answer,y,rm(x,print=T),
    ?answer,N,,
    ?answer,n,,
    ?(?MESSAGE(Say yes or no)?prmd(x)))
END
```

MACRO rmlist

```
?apply(prmd,$*)      # ask permission to remove list of names
END
```

MACRO cleanup(pattern/\$T*/)

```
?define(Tlist,encode(list("pattern"),",","$mac.Tlist") # list of all macros with pattern
?(?rmlist(?Tlist)) # next pass - ask for remove permission
END
```

Allows dynamic creation of a macro during execution of S expressions

The created macro can then be called by other macros, etc ...

Example: ?define(allnames,encode(list()),",",sep=""))

creates a macro "allnames" that contains the names

of all data sets on working storage

MACRO define(name)

?(\$mac.)name ← cstr(# create macro name with args as defn

 Data=c(\$*),

 junk=1 # needed to create structure

)

END

Reference

5. Reference.

This section describes the S language and system more formally. This is the place to find precise definitions of what is legal (syntax), what is the effect of expressions (semantics), and the full extent of the data structures and other system components. In addition, a number of specific features and details are introduced or elaborated.

5.1. The Language: Syntax.

The process of evaluating user's expressions in S proceeds in two phases: first, the expression as typed is *parsed* to produce a data structure; then this data structure is *interpreted* to evaluate the individual function calls in the expression. This subsection describes the parsing process, which in turn is done in two steps. User input goes to a *lexical analyzer*, a program which breaks the input into *tokens* (names, numbers, character strings, etc.). These tokens are passed to a *parser* which interprets the form of the expression as a whole. In the case of S, the parser operates on the basis of a set of *syntax rules*, specifying the form of all legal expressions. In fact, the rules are relatively compact and straightforward (this is one reason for the flexibility of the language). We will present in this section the *complete* syntax for S, with only a few alterations to enhance readability.

Each syntax rule specifies a *syntax type* and then enumerates all the possible forms which are recognized as producing this type. These forms are defined by concatenating three kinds of symbols: other syntactic types, literals (characters to be taken as themselves), and *token types*.

Lexical Analysis. A token type is some class of character strings which is recognized by a lexical analyzer, prior to parsing the S expression. The token types recognized in S are as follows.

INT an integer literal;

REAL a real (decimal fraction or exponential) literal.

NAME a string of characters, consisting of letters, digits, and ".", with the first character a letter. Depending on implementation, there may be a limit to the length of the name (on UNIX, for example, names must be unique in 14 characters).

STRING any string of characters enclosed in matching, unescaped double or single quotes.

OP a (binary) operator, one of the arithmetic or logical operators, ":", or a special operator consisting of "%" followed by any single character.

UNARY one of the permissible unary operators ("+", "-", "!").

MACRO "?" immediately preceding a name indicates a call to an S macro (see note (4) below).

Notes:

- (1) with the exception of STRING, no token can have internal white space, and, specifically, real numbers in exponent notation cannot have internal blanks;
- (2) tokens of type NAME are approximately the legal names of functions and data sets (with the respective exceptions of the existence of operators and the use of "prefix" -- both of which are discussed below);

- (3) the type *OP* is in fact subdivided, but this is only relevant for questions of *precedence* (see section 3.1.4);
- (4) the *MACRO* token will not be recognized inside a string or in a comment. When it is recognized, expression input is continued until a possible complete expression is obtained. The parse is then terminated and the text of the expression is given to the *macro* function. This function (if it does not encounter an error) expands all macro calls, and writes the expanded expression to a file. A hidden invocation of the *source* function then brings this file back in for parsing.
- (5) Comments in an S expression are recognized by the literal "#". The lexical analyzer then discards this character and the rest of the line. Thus, comments are not processed by the parser.

When the lexical analyzer finds a token, it returns both the *type* (a code for one of the types above), and also a token *value*, when appropriate, giving the actual text or data value of the token. Thus, "3.14159" is a token of type *REAL* and has the obvious value as a real data item. In the syntax rules of this section, it is the token type which appears in the definitions. When we talk about the semantics of the expressions, the token value will be used.

Syntax Rules. In the presentation of the S syntax, the syntactic type appears on the left of the rule, followed by ":" and the first possible form. Alternative forms are separated by a vertical bar character, "|". For clarity of presentation, names for syntactic types are lower case, names for token types are upper case, and literals are enclosed in quotes. Syntax rules are terminated by a semi-colon. Comments in the rules go from "#" to the end of the line.

The key syntax type in S is *expr*, the general S expression. Here is the rule which defines all legal expressions.

```

expr      : NAME "(" arg.list ")"           #function call
          | expr OP expr                   #binary operator
          | UNARY expr                     #unary operator
          | sub.expr "[" arg.list "]"      #subset
          | asn.expr "←" expr              #assign or replace
          | expr "→" asn.expr              #same, to the right
          | INT | REAL | STRING            #literals
          | sub.expr                      #can be subsetted
          | asn.expr                      #can receive assignments
          | control                       #iteration, conditional, etc.
          ;

```

These, with the exception of *control*, are the expressions encountered in most ordinary use of S (i.e., outside of macros). Clearly, they form a compact syntax, although very general expressions may be built out of the forms. The syntactic forms appearing on the right of the rules (in addition to *expr* used recursively) now must be defined. The syntactic type *asn.expr* is the class of things which can receive assignments: simple assignments or replacements of components and subsets.

```

asn.expr  : com.name                      # name or component
          | com.name "[" arg.list "]"     #subsetted
          ;

```

and *com.name* is a name, optionally followed by any number of named or numbered

components:

```
com.name : NAME
          | com.name "$" NAME
          | com.name "$" "[" expr "]"
          ;
```

The syntactic type *sub.expr* is the class of things of which subsets can be taken:

```
sub.expr : "(" expr ")"           # anything in parens
          | NAME "(" arg.list ")"  # function call
          | sub.expr "$" NAME      # component
          | sub.expr "$" "[" expr "]" # numbered component
```

However, the most common form of *sub.expr* is just NAME, which is not included in the rules above! This case is covered by the last line of the definition of *asn.expr*. To include the same form in the rule for *sub.expr* would be ambiguous.

The syntactic rule *control* provides conditional expressions, iteration and compound expressions:

```
control : "if" "(" expr ")" expr
          | "if" "(" expr ")" expr "else" expr
          | "for" "(" NAME "in" expr ")" expr           # iteration
          | "while" "(" expr ")" expr
          | "repeat" expr
          | "break" | "next"                             # loop control
          | "{" exp.list "}"                             # compound
```

The remaining syntactic forms *exp.list* and *arg.list* are lists of expressions separated by ";" and of arguments separated by ",".

```
exp.list : expr
          | exp.list ";" expr           # see note in §5.1.2
          ;
arg.list : arg
          | arg.list "," arg
          ;
```

While *expr* has been defined, *arg* has not. A function argument can be empty, or an expression, or a "name=" followed by an empty or an expression.

```
arg : # empty
     | expr
     | NAME "="
     | NAME "=" expr           # empty
     ;
```

This completes the essential rules. Now for some special cases, explanations and nits.

5.1.1. Function Calls and Commands.

As a concession to those who like a command-language appearance rather than a functional notation, function calls in the top-level S expression can use blanks, rather than parentheses, to separate the argument list from the function name. This is done by defining a syntactic type *command*:

```

command : expr                                     # as before
        | NAME BLANK arglist
        ;

```

The semantics of the second form is the same as that of the function call in the first line of the *expr* definition.

5.1.2. Compound Expressions.

The formal syntax implies that S insists on a semi-colon between expressions within a compound expression. Actually, the language is more liberal. A new line can be used to separate expressions, if the end of the line could be the end of an expression. This is accomplished outside of the parse, by the mechanism of returning ";" from the lexical analyzer when a new line occurs in this circumstance.

5.1.3. Continuation.

An interactive language interpreter must decide when an expression is complete, in order to execute it. (By contrast, a compiler can read a complete *program* before beginning the parse.) In S, the first legally complete expression will be parsed and executed. This becomes an issue in practice only when the user wants to type a new line, continuing the expression onto the next line. Continuation can be forced if either of the following holds:

- 1- the last token is not one of those which could legally end an expression;
- 2- there is an unbalanced "(" or "[".

The tokens which can end an expression are any literal (INT, REAL, NAME or STRING), ")", "]" or "}". To force continuation, manage to end the line with anything else. A special extension of the continuation rule is applied, however, so that conditional or iterative expressions can be written with the test on one line and the executed expression on the next. For example, after seeing

```
if(x>0)
```

the parser will expect a continuation, even though this would be a complete expression if "if" were a function name. The mechanism is to recognize the special tokens "for", "if" etc., and not signal parenthesis balance on reaching the ")". In complete conditional expressions, however, it is essential to get the "else" token in on the same line; otherwise a simple "if" expression is evaluated:

```
if( min(x)>0 )log(x) else
    abs(resid)
```

but not

```
if( min(x)>0 )log(x)
else abs(resid)
```

5.1.4. Reserved words.

Notice that several of the literal items in the syntax rules are also legal NAME tokens: "if", "else", "repeat", "while", "for", "in", "next", and "break". These can not be used as names; the lexical analyzer will recognize them first in their special sense. A syntax error will almost certainly follow.

5.2. Semantics.

This section describes what happens after an expression is parsed: how the S executive scans and evaluates the expression. An expression parsed according to the rules of section 5.1.1 becomes a hierarchical data structure. For example, a function call becomes a structure whose components are the structures resulting from parsing the arguments to the function. The S executive scans the structure and executes all the functions invoked, proceeding from the "lowest" level up. As each substructure is evaluated, the corresponding result (e.g., the data structure returned from a function call) replaces the parsed subexpression. When the entire structure has been scanned and evaluated, the parsed expression is replaced by its value.

The remainder of this section discusses the semantics of specific parts of the S language.

5.2.1. Functions and Operators.

The call to an S function in an expression can, naturally, only be evaluated after all the subexpressions appearing in the argument list of that call have been evaluated. The subexpressions in a given argument list are evaluated from left to right, a distinction which is relevant only when the subexpressions have side effects (e.g., assignment):

```
plot(xvar ← read(xfile), reg(xvar,y)$resid)
```

reads the file and assigns *xvar* before evaluating the regression. Expressions relying on the left-to-right evaluation are bad programming style, as their interpretation is usually obscure.

After the arguments are evaluated, the function call is a structure, whose *i*-th component is the data structure representing the evaluation of the *i*-th argument in the argument list. If the argument appeared in the "name=" form, then the component has a name which is the value of the corresponding NAME token. Otherwise the component has a null name. Notice that the NAME token in this case turns into the name of a structure component: it has nothing to do with the name of either a dataset or a function.

After the arguments are evaluated, the executive searches for a function with the desired name. The default search list is the set of functions included in the local S implementation. If one or more calls to the *chapter* function have occurred, each named chapter is placed on the search list (by default, on the front of the list). Since the search terminates as soon as a function of the given name has been found, one can redefine standard functions by putting a function with the same name on a user's chapter.

Once the function name has been found, the S executive transfers control to the corresponding function, giving it the *argument structure*; i.e., the data structure representing the evaluated form of the actual arguments to the function. The processing of the function arguments and all further interpretation is the job of the individual function. The S executive has no knowledge of how many or what kinds of arguments are required by any functions.

When the function has completed execution, it returns its result as a data structure to the executive, provided execution terminated normally. The result replaces the call to the function in the parsed expression and evaluation continues. If an error or user interrupt occurred during the function execution, the execution of the entire

current expression is stopped, the current expression may be written to the file "sdump" for use by the *edit* function, and the executive attempts to get the next expression from the user.

Argument processing is entirely the choice of the individual function. Since argument processing is interpretive, anything is possible and the actual variety is great. However, there is a simple form of argument matching that is used by most functions. Details are given in section 6; we give here the rules as they affect use of the S language. The function has a set of formal arguments, with the names and the order implied by the detailed documentation for the specific function. For example, the *regress* function has full calling sequence in the following form:

```
regress(x, y, wt, int, print, names, q)
```

The formal arguments are matched to the argument structure: for each formal argument, in order, a partial-match algorithm is applied. This scans each component of the argument structure. If the name of the component matches the name of the formal argument exactly, the two are matched, the search is terminated and the component is marked USED (so it cannot match any other arguments). If no exact match occurs in the complete argument structure, the argument may be matched in two other ways:

- (1) there is exactly one *partial match* of names, where currently this means that the component name is a leading substring of the formal name;
- (2) there is no partial match, but a component has been found with a null name; i.e., not specified in the *name=value* form and not USED. The *first* such component is matched.

In either of the cases, the component is matched to the formal argument and is marked USED. In all other circumstances, the formal argument is marked MISSING. An argument may also be made explicitly MISSING by an empty actual argument, either positionally or in the *name=* form.

After the matching process, the S function will try to *coerce* the actual argument to the data type and mode desired, if the argument is present, or will take the chosen default action, if it is MISSING. If the argument is MISSING and not optional, or if the coerce fails, an error is generated and the expression is aborted.

When all the formal arguments have been processed, the argument structure is scanned for any remaining (non-USED) components. If there are any, an error is generated. Notice that this usually ensures that misspelled argument names will be detected, even if the corresponding argument was optional. (The partial match could, but currently does not, attempt to correct misspellings and/or resolve multiple partial matches.)

Here are two examples of calls to the function *regress(x,y,wt,int,names,q)*:

```
regress( cbind(age,iq,height), income )
regress( xmat, abs(resid), y=yvar, in=FALSE )
```

In the first example, the function *cbind* is evaluated first; the data structure representing the result is an (unnamed) component of the arguments to *regress*. There will be no exact or partial match for "x"; the result of *cbind* will match positionally. Similarly, *income* will match "y" positionally. All remaining arguments will be MISSING. In the second example, "x" will again be matched positionally, this time to the data structure resulting from getting *xmat* from a database. However, "y" is now matched by name; it does not match *abs(resid)* which will, at the next stage, match "wt" positionally. The

argument named "in" matched "int" partially; since there is no exact match, this succeeds.

From this standard form of argument processing, there are many variations. Some of the more important are as follows:

- A number of functions (e.g., *c*, *print*, *save* and *rm*) take arbitrarily many arguments. In this case, the actual arguments may still be named, but in general there will be no name matching (with the exception below). The function has access to the argument names, which it can use if appropriate (see *save*, e.g.).
- Arguments which appear in the form *name*= in the documentation may appear only by name and will never match an unnamed actual argument. Such arguments allow for special flags to functions, like *print*, that take arbitrarily many arguments.
- The test for extra or misspelled arguments is suppressed in some functions. The most common reason is that the function will later *chain* to another function, passing on any unused arguments. For example, most high-level graphics functions chain to *title* to process the arguments *main*, *sub*, *xlabel*, and *ylabel*. Misspelled arguments will be detected, but not by the function originally called.

To repeat, specific functions can interpret arguments as they wish; the examples given are just the most common.

Operators. Semantically, there is no distinction between operators and general functions in S. Syntactically, operators cannot have named or MISSING arguments, and the rules for names of functions and names of operators are mutually exclusive, but these distinctions have no effect on the way the corresponding functions are executed.

5.2.2. Side Effects: Database Changes and Parameters.

The semantic effect of most S functions is as described above. The substructure representing the call to the function is interpreted, and the result replaces it in the structure representing the entire expression. There are just two ways in which functions can have additional semantic effect: by altering the contents of one of the S databases or by changing the value of some of the internal S parameters.

Database contents may be changed by functions which create or remove datasets (*assign*, *save*, *define*, *restore* and *rm*) or which modify existing datasets (*edit*, *medit*). The function *assign* corresponds to all forms of assignment operator: "<—", underline and "—>". The *asn.expr* syntactic type will be turned into a sequence of arguments to *assign* which are either character strings, corresponding to names of datasets or components, or integers providing component numbers. The syntax of the assignment expressions restrict the dataset and component names (see the documentation for *assign*). The function *save* is similar to assignment, except that it takes multiple datasets and does not replace components or subsets of existing datasets. For each argument, *save* will use the name, if any, of the corresponding component of the argument structure as the name of the saved dataset. If there is no specified name, the argument data structure must itself have a name. The arguments given without a name can only be datasets from other databases. Thus, *save(newx=sqrt(x))* and *save(x)* will work, but *save(sqrt(x))* will not.

The function *define* chains to *save* to store the macro definition. The macro name, extracted from the definition for each macro, becomes the dataset name of each of the macros, with "\$mac." prepended. The function *edit* either chains to *assign*

or creates a hidden call to *source*, if called without argument. The function *medit*, on the other hand, chains to *define*, so that references to macro arguments by name in the edited text can be converted.

Parameters internal to the S system may be set by the *options* function, by any of the functions which alter search paths for functions or databases (i.e., *chapter*, *attach* and *detach*), or by *par*, which sets graphical parameters. All such parameter settings hold until reset or until the user quits from the session. They are initialized to default values when the user logs into S. Graphical parameters are also reset to default values when the user specifies a new graphics device. Thus

```
hp72h; par(pch="+")
```

```
...
```

```
hp72v
```

leaves the plotting-character parameter, *pch*, with its default value of *"*"*.

5.2.3. Compound Expressions.

A compound expression, consisting of subexpressions separated by semicolons or new lines and enclosed in braces, is parsed into a structure similar to an argument list. The components of the structure are the parsed subexpressions.

The important aspects of compound expressions (particularly for understanding conditional and iterative expressions) involve the order of calculations in evaluating the expression and the rules for the value returned. With the exception of *break* and *next*, the subexpressions are evaluated in order. When a subexpression has been evaluated, its value becomes the *current value* for the compound expression. However, it does not replace the parsed form of the subexpression in the compound expression (since the expression may be iterated). The final value of the compound expression is its current value at the time when the S executive finishes execution of the expression. This may occur from encountering the closing *"}"* of an uniterated compound expression, from execution of a *break* or *next* in the compound expression, or from the iteration tests of *for* and *while* expressions. In interactive processing, user interrupts can also be used to terminate execution of a compound expression. Three forms of subexpression do *not* alter the current value of the compound expression: *break*, *next*, and a simple *if* expression (no *else* expression) whose test returns FALSE.

A frequent confusion in practice is the assumption that subexpressions in compound expressions are automatically printed. They are not, because the automatic printing mechanism only operates at the recognition of the *command* syntactic type; i.e., at the highest expression level. To force printing of subexpressions, these must be passed to the *print* function as arguments:

```
for(i in list())  
  print(i,get(i))
```

prints the names and values of all datasets on the working database.

5.2.4. Conditional Expressions; Iterative Expressions.

The general conditional expression

```
"if" "(" expr1 ")" expr2 "else" expr3
```

is evaluated as follows: *expr₁* is evaluated and coerced to a logical vector. The first data value in the result is tested. If it is TRUE, then *expr₂* is evaluated and its value

becomes the value of the conditional expression. If the tested value is FALSE, and no *else* expression was given, the value of the conditional expression is undefined. Otherwise, $expr_3$ is evaluated and its value becomes the value of the conditional expression.

Of the three types of iterated expressions, the evaluation of the the expression

"repeat" $expr$

is a model for the others. The repeated expression is evaluated repeatedly until the occurrence of a *break* in $expr$ terminates iteration, at which time the value of the *repeat* expression is the current value of $expr$. Note that a *break* expression causes a jump out of the nearest surrounding compound expression. Similarly, a *next* expression is equivalent to a jump to the point just after the last subexpression in the nearest surrounding compound expression.

Evaluation of the expression

"while" "(" $expr_1$ ")" $expr_2$

is identical to a *repeat* expression, except that before each evaluation of $expr_2$, the parsed form of $expr_1$ is evaluated, coerced to LOGICAL, and the first data value of the result is tested. If it is TRUE, $expr_2$ is evaluated; otherwise, the iteration terminates and the current value of $expr_2$ (if any) becomes the value of the *while* expression. In evaluating

"for" "(" NAME "in" $expr_1$ ")" $expr_2$

$expr_1$ is first evaluated and coerced to a vector of any mode. Then the value of the token NAME is assigned to an internal dataset which is a vector of length 1 and the same mode as $expr_1$. The data value of this internal dataset is set to the first element of $expr_1$ before the first evaluation of $expr_2$, to the second element before the second evaluation, etc. Evaluation of $expr_2$ continues until there are no more elements in $expr_1$ or until a *break* occurs.

Notice that the dataset used is *internal* (see the discussion of names and datasets below), rather than the result of an assignment. This means that the *for* loop has no side effect on any database. After execution of the expression terminates the internal dataset is removed. During the evaluation of the compound expression, the internal dataset hides any dataset with the same name on one of the databases. It is even possible to re-use the same internal name in several nested loops (which could occur if the loops were set up in different contexts, e.g. inside macros).

5.3. Data Structures.

The general S data structure can be defined recursively as follows. A data structure is a list of the form:

(name mode length value₁ value₂ ...)

where:

name is a character string, in quotes, defining the name of the structure;

mode is one of the vector modes (LOGICAL, INTEGER, REAL, CHARACTER) or else is of the form

STRUCTURE type

indicating a hierarchical structure of special type *type*. an integer providing a

quick means of identifying particular forms of hierarchical data structures, e.g., arrays, time-series. In the *dget* and *dput* functions, modes are abbreviated to single characters ("L", "I", etc.).

length is the number of values (for a vector) or components (for a structure) and it is understood that there are exactly *length* occurrences of *value*_{*i*}. In the case of mode STRUCTURE, the individual values will themselves be lists of this form (vectors or further structures).

For example, the matrix defined by

```
> x←matrix(1:12,3,4)
```

would have the form

```
( "x" S 21 2
  ( "Dim" I 2 3 4 )
  ( "Data" I 12 1 2 3 4 5 6 7 8 9 10 11 12 )
)
```

(the integer 21 is the internal type for arrays).

Data may be written to a file in this form by the function *dput*. Conversely, a file containing a data structure in this form can be read into S by the function *dget*. The related functions *dump* and *restore* write and read lists of datasets. In addition the *extract* macro and utility function extract datasets from data files consisting of identically formatted records. These functions provide conversions between internal S data structures or user files and a general list representation of data structures. In particular, notice that this list form of representation for data structures is both portable and completely general. All possible S data structures can be represented in this form, without loss of information.

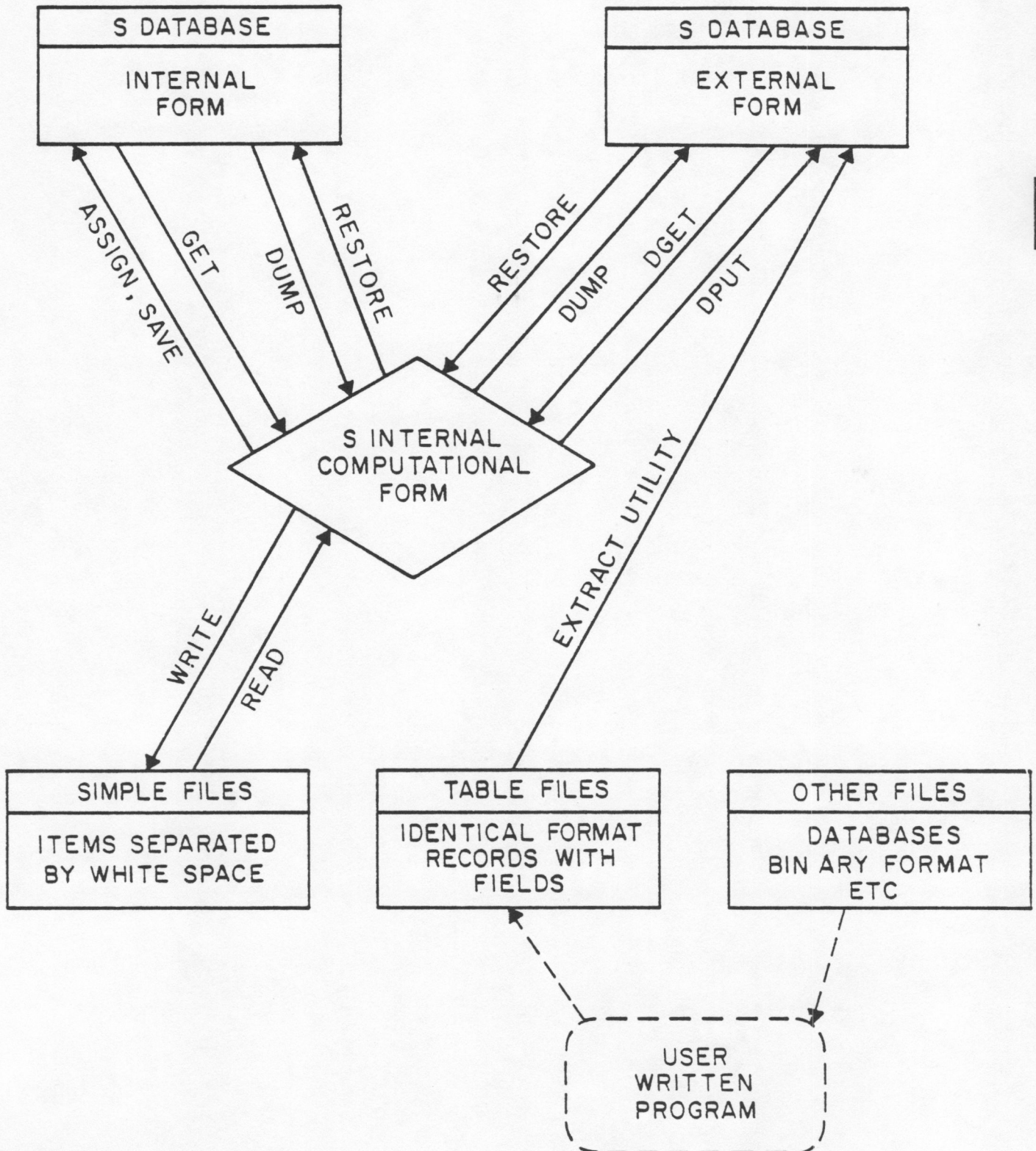
The list form shown here is the *external* representation of S data structures. There is also an *internal* representation, solely for reasons of efficiency. The S user should never need to know about the internal representation. Essentially, internal representation is used instead of characters to represent logical, integer and real data items, and some pointers are used to find data values. It is important, however, to understand how the different external and internal forms relate to S functions that manipulate data:

- the functions *read* and *write* transform between data items on files, without any structural information, and vectors in S;
- the functions *dget* and *dput* transform between data structures on files in fully general external form, and the corresponding structure in an S expression;
- the *get* and *assign* functions transform between datasets on an S database (in internal form) and the corresponding data structure in an S expression.

In most cases, data which was generated outside S will be accessed either through the *read* function or through the *extract* macro or utility function. Transmission of data, particularly large numbers of datasets, between S databases on different machines will be accomplished by *dget*, *dput*, *dump* and *restore*.

The relationships among files of various forms, S datasets and functions that convert from one form to another is shown symbolically in the following figure.

DATA MANIPULATION



Writing New Functions for S

6. Writing New Functions for S.

The macro facility described in section 4 allows users to extend the S language by defining macros to carry out specialized computations, usually depending on one or more arguments. Eventually the macros must expand into expressions in the S language. This somewhat limits their efficiency and generality. A different way to extend the language is to write *new* functions in S which typically make use of algorithms (subprograms) which carry out some computational task. As new methods, improved algorithms and new areas of application arise, new functions incorporate the advances into the S language. The key to the ability to define new functions simply is the *interface language*. Each S function is defined by an interface routine, written in the special-purpose interface language. When an S user invokes the function, the interface program is responsible for interpreting the arguments to the function and for returning the result of the function.

The computational job of the function is usually carried out by algorithms called from the interface routine. In this section, we will usually treat the underlying algorithms as given. They may be FORTRAN routines taken from a subroutine library or they may have been written for some other purpose, perhaps outside of S. In fact, S provides an environment of software support for the writing of such algorithms. Section 7 will describe this environment, which centers on an *algorithm language*.

Macros and new functions are both useful ways to extend the S language, but are useful in different circumstances. Macros build on the language by condensing complicated expressions, which recur in similar forms, into a single, simpler macro call. The user of the macro is relieved of the details of the underlying S expressions. However, macros can only produce computations in terms of existing S functions. When the computation involved is a reasonable use of S, macros have the advantages of being relatively simple to define and easy to understand.

Sometimes, however, the computation desired can only be defined in a clumsy or highly inefficient way in S. This is particularly true if the computation requires iterating over single elements of the data. Also, the underlying computation may be already available in the form of a published algorithm or other subroutine. In these cases, the definition of a new S function, through writing an interface routine, may be the best approach. Writing the interface routine and getting it and the underlying algorithms to work involves a greater initial investment of time than creating a macro, however. It also assumes some experience on the writer's part with programming languages. Specifically, both the interface language and the *algorithmic language* to be described in section 7 use RATFOR and the FORTRAN language as intermediate forms. Some familiarity with these is desirable, before one begins writing interface routines (e.g., *Unix Documentation*, Vol. 2, Section 22).

The essential principle in writing new S functions is: KEEP IT SIMPLE. For initial efforts, particularly, a clearly defined, well understood function will give best results, even though it may seem to lack some of the generality possible. It is more productive to use and adapt a simple function in the interactive environment of S than to spend time trying to debug an overly elaborate, poorly understood interface routine or algorithm.

6.1. Design and Implementation of Simple S Functions.

6.1.1. Design of S Functions.

Before doing any actual programming, one should have a fairly clear idea of the purpose and form of the new S function. What input information should it use (arguments), what output information should it provide (results) and what computations must it do in between? If these are understood in general terms, and if they are clear and well defined, the implementation of the ideas in a new S function will be much easier.

Throughout section 6.1 we will consider a simple but typical example of an S function. Suppose we wish to write a function to fit a linear model to a set of data, given some existing algorithm(s) which carry out the desired numerical computations.

Suppose we decide to call the function *fit*. Without worrying about details, here are some design decisions needed.

Arguments We decide to fit a single variable to several other variables. As usual in S, the predicting variables are assumed to be columns of a matrix. We should check that the arguments make sense (for example, that they correspond to the same number of observations); otherwise the algorithm may return nonsense or die mysteriously.

Results A linear model, once computed, could be represented just by the coefficients computed. However, it is usually more convenient for the user and more accurate computationally to compute the residuals as well. All the S functions treating linear models provide this information.

Options Initially, we should try not to provide too many options. As experience with the new function grows, the important variations will become clearer. For linear models, two common options are: whether the model should include a constant (intercept) term; and the provision of weights for the observations (assuming an algorithm exists which handles weights).

Algorithms We need to consider the algorithm(s) available, both to decide whether they are suitable as they stand and also to make sure we provide all their requirements. For example, many computational algorithms require some working (scratch) storage. The interface routine provides this.

The remainder of section 6.1 illustrates how the interface language provides these facilities. In the process, we will develop an interface routine for *fit*.

6.1.2. Arguments; the FUNCTION Statement.

The first statement of an interface routine is of the form:

```
FUNCTION name( arg1, arg2, ... )
```

where *name* is the name of the S function and *arg1*, etc. are the descriptions of its arguments. Each argument is defined by the name of the argument, followed by a *description list*:

```
name/ spec1, spec2, ... /
```

The description lists are used in essentially the same form both for arguments and for dynamically allocated data structures (section 6.1.4). For arguments, the specifiers *spec1*, etc. give the desired type (MATRIX, TS, etc.) and mode (REAL, LOGICAL, etc.) for the arguments. We describe the process of turning the actual arguments into

the desired type and mode as *coercing*. Details of the process will be introduced as we encounter more specialized needs. In the simplest form of *fit*, we want a vector of values and a matrix to predict the vector. These are usually named *y* and *x* in S functions:

```
FUNCTION fit( x/MATRIX/, y/REAL/ )
```

The example shows that some abbreviated specifications are possible. Those of most frequent use are:

- if the structure type is omitted, it is assumed to be a vector structure (section 3.2.1), so that the argument may be treated as a simple vector;
- if the type is given (e.g., MATRIX) but the mode is not, the default mode is REAL.

When an argument is optional, the corresponding description list may just say OPTIONAL, or may specify the default data structure to be allocated when the argument is missing. In the second case, the description is the same as would be used to allocate the structure dynamically (see section 6.1.4). In either case, the interface routine can make use of the logical expression *MISSING(name)* to test later whether the argument was actually missing. In the *fit* example, suppose that an argument, *wt*, provides optional weights for use in the fitting. If there exist two algorithms, one with and one without weights, the interface routine can simply check whether weights were provided and call the corresponding algorithm. We just need to make the argument *wt* optional, without providing a default data structure.

It is possible, also, to have an optional argument, *int*, to control whether the model should include an intercept term. In this case, the default action to take is simply to interpret the argument as TRUE. This can be built into the argument declaration, by specifying the default structure to be of length 1 and value TRUE:

```
FUNCTION fit( x/MATRIX/,
              y/REAL/,
              wt/REAL,OPTIONAL/
              int/LGL,1,TRUE/ )
```

We see also in the above that the FUNCTION statement can and should be spread out over several lines, if this improves readability.

6.1.3. Error Checking.

The interface language generates error messages automatically if the actual arguments (given by the user) do not match the arguments in the FUNCTION statement (too few, too many, or the wrong names), and also if the actual arguments can not be coerced to the specifications given. For simple functions no further checks may be required. The interface routine should check explicitly if the function requires that lengths be consistent or that data values fall in a specific range, or for any other substantive restrictions. If any requirements are not met, the interface routine should issue a (meaningful) error message. An error message in an interactive environment gives the user a chance to correct the expression and to try again. There is no need to go to great lengths to produce a result from bad input data.

The errors are checked by computing expressions involving the arguments and checking that the computed result satisfies requirements for use in the function. If not, an error message is printed and execution of the S expression is terminated by

FATAL(message)

The computations for the test usually involve attributes of the argument data structures, such as **LENGTH**, **NROW**, **NCOL**, and **MISSING**.

In the *fit* example, one would require that the number of rows in the *x* matrix and the lengths of *y* and *wt* match. Failure of these requirements should generate an error and a message to the user through **FATAL**:

```
if(NROW(x)!=LENGTH(y))
  FATAL(Number of observations in x and y must match)
if(!MISSING(wt) & NROW(x)!=LENGTH(wt))
  FATAL(Number of weights must match number of observations)
```

Only the facts go in the argument to **FATAL**. The name of the function is added to the message automatically.

6.1.4. Dynamic and Static Data Structures.

The arguments to the S function and the results returned from it must be dynamically allocated, self-defining S data structures. Structures which are arguments may also be returned as results. Otherwise, it is necessary to allocate the results by declaring them as dynamic structures:

```
STRUCTURE( arg1, arg2, ... )
```

where, as with the **FUNCTION** statement, *arg1*, etc., are of the form:

```
name/ spec1, spec2, ... /
```

In the **STRUCTURE** statement, the specifications must define the actual length or similar description of the structure, as well as the type and mode. These actual parameters are given by a computational expression which may depend on attributes of other structures; these parameters do not need to be constants. In the *fit* function,

```
ncoef=NCOL(x)
if(int)ncoef=ncoef+1 #for the intercept
STRUCTURE( coef/REAL,ncoef/, resid/REAL,LENGTH(y)/ )
```

allocates vector structures *coef* and *resid* based on attributes of the arguments *x* and *y*. Notice that the attributes, like the length of *coef*, are expressions computed at run time.

Allocated structures need not be results of the function. If the algorithms used require scratch space of variable size, this can also be conveniently allocated through a **STRUCTURE** statement.

When the allocated structure is a matrix, two actual parameters need to be provided, the number of rows and the number of columns:

```
STRUCTURE( xtrans/MATRIX,NCOL(x),NROW(x)/ )
```

Notice that the order of the parameters determines their interpretation. For time-series structures, three parameters are required: the start time, end time and number of values per year.

An alternative specification is frequently useful: when an allocated structure should be identical to an existing structure, the short form

LIKE(name)

may be used in place of any other specification. In the *fit* example, a better specification of the *resid* structure would be:

```
STRUCTURE( resid/LIKE(y)/ )
```

Not only is this more compact to write, but it ensures that *resid* inherits all the attributes of *y*; for example, if *y* is a time-series, the residuals will be also. This facility is not necessary for producing scratch copies of arguments to S functions: arguments are already dynamic data structures and can be overwritten without harming user datasets.

Data values may be specified as an additional parameter to the **STRUCTURE** specifications, or may be entered later.

Declarations for scalars, functions and other non-S data are identical to similar declarations in RATFOR or FORTRAN, except that these are **STATIC** data items in the interface language. The whole FORTRAN-style declaration must be included in a **STATIC** statement; e.g.,

```
STATIC( real xlim(2),mean; integer i,j )
```

Similarly, FORTRAN common block declarations should be included in a **STATIC** statement. Otherwise, the rules about what needs to be declared and how are identical to RATFOR and FORTRAN.

Attributes. All S dynamic data structures are self-describing. This means that their type, mode, length, etc. are available at execution time by examining the structure. The interface language provides a number of attribute expressions to help in this process. The full set will be discussed in section 6.2. Some commonly useful attributes are

```
MISSING(x) # was it supplied
```

```
LENGTH(x)
```

```
NROW(x), NCOL(x) # number of rows, columns
```

```
TSTART(x),TEND(x),TNPER(x) # time-series parameters
```

Expressions to access individual data values of vectors, matrices, etc. may be specified as in the S language:

```
x[i,j]
```

```
y[i-1]
```

but within the interface language it is not possible to refer to subsets of more than one item in this way.

All other uses of the name of the data structure by itself (for example, in a call to a subroutine) are interpreted as a reference to the data values (see the examples in 6.1.5).

6.1.5. Computations.

The interface language contains the full syntax of RATFOR to express computations. However, computations for S functions should primarily be carried out in algorithms called from the interface routine. Keeping to this approach makes it likely that the algorithms will be useful more generally, and usually increases the efficiency of the resulting S function. Subroutines are called just as in the algorithm language itself, with the added features of reference to S data structures and their attributes as

illustrated above.

Suppose the *fit* function calls a subroutine *lsfit*:

```
subroutine lsfit( x, nr, nc, y, coef, resid)
real x(nr,nc), y(nr), coef(nc), resid(nr)
...
```

The interface routine calls this subroutine, using the arguments to the S function and their attributes to define the information needed by the subroutine:

```
call lsfit(x,NROW(x),NCOL(x),y,coef,resid)
```

When the algorithm *lsfit* has finished its computations, the data values in *coef* and *resid* are the desired coefficients and residuals. Now the S function *fit* wants to return a structure with components named *coef* and *resid*, which is done by the RETURN statement in the interface language:

```
RETURN(coef,resid)
```

Then the complete *fit* interface routine for a simple version of the function could look like:

```
FUNCTION fit( x/MATRIX/,
             y/REAL/ )
if(NROW(x) != LENGTH(y))
    FATAL(Number of observations in x and y must match)
STRUCTURE( coef/REAL,NROW(x)/, resid/LIKE(y)/ )
call lsfit(x,NROW(x),NCOL(x),y,coef,resid)
RETURN(coef,resid)
END
```

6.1.6. Creating New Functions.

At this point, enough of the interface language has been described to write an interface routine corresponding to a simple S function. It remains to take the interface routine, along with any other routines needed, and turn them into an executable S function. There are a set of utility commands which do this. They are executed at the shell (operating system) level.

In order to develop a set of S functions, one must create a private *chapter* (essentially, some directory space in which to put the functions and their documentation), through the utility

```
S CHAPTER
```

This is used only once to create the chapter. A new function within the chapter is announced once by the command

```
S FUNCTION name file1 file2 ...
```

with *name* the name of the function as called in S, and *file1*, etc. the names of files containing the interface routine and any other routines (algorithms) on which the function depends. The interface routine should be written to a file with a name ending in ".i". Routines in RATFOR, FORTRAN and C should be written to files with names ending in ".r", ".f" and ".c" respectively. Routines in different languages should not be included on the same file, since different processing is done by the S utilities for different languages. In the example of the *fit* function, if the interface function is

stored on file *fit.i* and the (RATFOR-based) source for the algorithm is on *lsfit.r*, the function is announced by:

```
S FUNCTION fit fit.i lsfit.r
```

Note that the FUNCTION utility determines the name by which the function can be called within S. This will generally match the name in the FUNCTION statement. The FUNCTION utility is used just once for each function. Special options are possible for the function; we will discuss these when we come to the S functions that need them.

Once the function has been announced and the source code written for the interface routine and other routines, the function can be created by

```
S MAKE name
```

This initiates a number of steps to compile the interface routine and other routines needed and to load the executable version of the function. As the name of the utility suggests, this is a version of the *make* program (Feldman, 1978, in Volume 2 of UNIX documentation). The FUNCTION utility, in fact, creates rules by which *make* can create the executable form of the function. Because of special rules used in making S functions, however, the S MAKE utility must be used.

Once the function has been made, it can be tested from within S. The S function *chapter* is used to put the user's chapter on the search list for functions. Specifically, if the chapter was created in the same directory used to run S, the S command

```
> chapter
```

can be used without arguments. Otherwise, the argument to *chapter* must be the name of the directory in which the chapter was created:

```
> chapter("/usr/abc")
```

(The UNIX command *pwd* gives the full pathname of the current directory.) Also, the function

```
search(1)
```

gives the list of chapters currently being searched by S, including the standard S chapter.

After testing, changes may be made in the interface routine or in other routines. Whenever changes are made, the function must be brought up to date by repeating

```
S MAKE name
```

Only if a function is revised to depend on a different set of files, should the FUNCTION command be used again. For example, if the *fit* function is revised to allow input weights and now needs to additionally call the algorithm *lswfit*,

```
S FUNCTION fit fit.i lsfit.r lswfit.r
```

Documentation Creating a file of documentation for the function is aided by the utility

```
S PROMPT file.i ...
```

that constructs partial documentation files (named "*file.d*") for functions. These files contain embedded instructions preceded by the character "~", that describe what should

appear at that point in the documentation file. The documentation file should be edited to contain reasonable descriptions of the arguments, results and other significant properties of the function.

Once the documentation file has been prepared, it is entered into the chapter documentation by

```
S NEWDOC file.d ...
```

After NEWDOC is used, the documentation file can be removed. Any user searching the chapter will now have access to the documentation via *help*.

To make changes in the documentation once it has been installed in the chapter, use

```
S EDITDOC -f file
```

which will recreate "*file.d*". This file can be edited, and then re-installed by NEWDOC.

Function documentation can be printed on the user's terminal by

```
S PRINTDOC -f [name ...]
```

All documentation for the chapter is printed unless the list of names appears. Altering the flag "-f" to "-ft" produces phototypeset documentation; "-fo" does the printing on an offline printer.

6.1.7. Hints; Debugging.

The fundamental maxim remains: KEEP IT SIMPLE. Try to start with an uncomplicated interface routine and a straightforward set of underlying algorithms. Let the rest of the S language do the work of reading in information, printing results and organizing data. In particular, it is not a good idea at this stage to print anything more complicated than the error messages shown above. Still less desirable is to try to read data values from a file.

S has facilities, however, for printing debugging information, either from an interface routine or from an underlying RATFOR algorithm. The statement

```
DEBUG
```

in either case invokes, during the execution of the S function, an interactive debugging facility. The user responds by typing the name of an S structure, as it appeared in the interface routine. The current contents of the structure will then be typed on the user's terminal, in the format used by the S function *dput*. (This is not as pretty as S output, but is more general and is adequate to see the current values in the structure.) To exit from DEBUG, type an empty line. You can interrupt the printing of a dataset; DEBUG will return to prompt for more input.

Notice that the name typed to DEBUG is the *keyname*, the name which appears in the FUNCTION or STRUCTURE statement, not the name of the actual data set supplied. To see a table of the available dataset names, type "?" to DEBUG. The response will be a table of keynames, datanames and an indication of whether the corresponding dataset was MISSING in the actual call.

In the *fit* function, suppose the DEBUG statement is inserted just in front of the call to *lsfit*. Then a session in the use of the function might be

```
> fit(pred[,1:2],response)
```

```
Debug called from fit:
```

```
Debug:?
```

Keyname	Dataname	Missing?
x		F
y	response	F
coef		F
resid		F

```
Debug:x y
```

```
x:
```

```
( "" S 21 2
```

```
( "Dim" I 2 16 2 )
```

```
( "Data" R 32 83 88.5 88.2 89.5 96.2 98.1 99
```

```
100 101.2 104.6 108.4 110.8 112.6
```

```
114.2 115.7 116.9 234.289 259.426 258.054 284.599
```

```
328.975 346.999 365.385 363.112 397.469
```

```
419.18 442.769 444.546 482.704 502.601 518.173 554.894 )
```

```
)
```

```
y:
```

```
( "response" S 0 1
```

```
( "" R 16 60.323 61.122 60.171 61.187 63.221 63.639
```

```
64.989 63.761 66.019 67.857 68.169
```

```
66.513 68.655 69.564 69.331 70.551 )
```

```
)
```

Even when invoked from the RATFOR routine, the interactive DEBUG feature dumps only the S data structures, as known in the interface routine. Since these are generally the data passed down to algorithms, however, one can monitor the computations by knowing how the algorithms were called.

In either interface routines or RATFOR routines, it is also possible to use DEBUG in a non-interactive way to print out single datasets. For an S dataset, say

```
DEBUG(name)
```

with *name* again the keyname of the S structure. For non-S data, such as variables in a RATFOR routine,

```
DEBUG(data,mode,length)
```

where *data* is some expression, *mode* is one of the data modes (REAL, INT, LGL, CHAR) and *length* is the number of values to print out.

The use of DEBUG, plus careful planning and a gradual approach to introducing complexities should make the debugging of new functions relatively straightforward. It is possible to run underlying algorithms with a stand-alone test routine (section 7), but with a little experience most users should find it more convenient to use the flexibility of S for testing.

6.2. Data Structures.

This section discusses in detail the data structures which can be defined in the interface language.

6.2.1. Data Values and Attributes.

Good use of the interface language should leave most of the actual computations on data to underlying algorithms. Occasionally, it may be more convenient to do some computation in the interface routine, particularly when working out default actions. Individual data elements on S data structures are given in bracket notation, as in S:

```
x[i,j] = 1
amax1(y[i],0.)
```

Since the references must eventually be translated into FORTRAN, only single values can be worked with. When calling subroutines, column j of the matrix x is passed as $x[l,j]$ and row i as $x[i,1]$ (of course, individual data items in a row are spaced $NCOL(x)$ apart).

Since all datasets are self-describing, they carry *attributes* as well as data; i.e., those values which define the structure in S. Vectors and vector structures (including matrices, arrays and time-series) will have defined the attributes:

```
MODE(x)
LENGTH(x)
```

In addition, matrices will have defined $NROW(x)$ and $NCOL(x)$ for the number of rows and columns. Time-series will have the start, end and number of observations per time period defined as the (real valued) attributes $TSTART(x)$, $TEND(x)$ and $TNPER(x)$. The function *diff* defined below, shows the use of time-series attributes and also an example of computations which are actually simpler in the interface language, since they largely involve manipulation of attributes.

```
FUNCTION diff(x/TS/, lag/INT,1,1/ )

STATIC( real start; integer i)
if(lag > LENGTH(x) | lag < 1)
    FATAL(Number of lags not between 1 and length of data)
start = TSTART(x) + float(lag)/TNPER(x)
STRUCTURE(y/TS,start,TEND(x),TNPER(x)/)

do i = 1, LENGTH(x) - lag
    y[i] = x[i + lag] - x[i]

RETURN(y)
END
```

6.2.2. Character Data and Character Attributes.

Computations on real, integer and logical data values map down to the underlying algorithm language with little change. Computations with character string data, however, require some additional discussion, since FORTRAN has only limited facilities to deal with character data.

Data in S which is of type character consists of *pointers* to character strings; for example, a vector of mode CHARACTER is a vector of pointers, each of which points to some text. The text for each character data item is terminated by the special character EOS. As a general rule, interface routines pass around pointers to the text. When it is necessary to process the text itself (as, for example, when printing it), the TEXT attribute gets the actual text. For example, if *labels* is an argument of mode CHARACTER, then

```
labels[i]
```

is the (pointer to) the *i*-th string, and

```
TEXT(labels[i])
```

is the actual text. The two most common examples of the use of character strings are in printing information and in calls to graphical algorithms (sections 7.2 and 8). A third example is the use of names for options to functions (e.g., choice of one of several possible methods). This also brings in the S technique for matching names, discussed in section 6.3.

Default values can be generated for character string data. The simplest case is that of a single character string, such as the title for a plot or printing. When this is an optional argument, a default pointer to a chosen string can be allocated by the attribute STRING:

```
FUNCTION qqnorm(
...
  xlab/CHAR,1,STRING(Normal Quantiles)/
...
)
```

The argument *xlab* defaults to a character vector of length 1, pointing to the string "Normal Quantiles". When the character vector has several values, one needs to use the encoding macros or the CTABLE statement to create a list of values. Section 6.3 contains an example.

It is possible to declare and initialize a table of character strings; say, to be used to look up possible values of a character-string argument. The interface language statement:

```
CTABLE(table,entry1, ..., entryk )
```

initializes *table* as a (STATIC) vector of pointers to character strings of length *k*. The first element of *table* points to a character string containing *entry₁*, etc. For example,

```
CTABLE(meths,average,connected,compact)
```

creates *meths* of length 3 containing pointers to the strings "average", "connected" and "compact". It is possible to look up argument values in such a table using the same partial match strategy used to match S argument names (see section 6.3.1). This is done by the algorithm *match*.


```

FUNCTION clust( ...
    method/CHAR,1,STRING(compact)/, ...

CTABLE(meths,average,connected,compact)
i=match(method,meths)
if(i<0)FATAL(bad value for method)

```

The value returned by *match* will be -1 if no successful match was possible, and otherwise will identify which value in the table matched. Note that the arguments to *match* are the pointer to the actual value and the vector of pointers defining the table. The CTABLE statment marks the end of the table by a pointer having the special value USED. It is possible to construct such tables by any method desired, although the CTABLE statement is typically the simplest and most convenient.

Certain character pointers are defined as attributes in the interface language. For example, FNAME is a pointer to the name of the S function. For each data structure in the interface routine, KEYNAME(x) is a pointer to its structure keyname and DATANAME(x) is a pointer to the name of the actual argument, if it had one. Thus, in the use of DEBUG in section 6.1.7, the keyname of *y* was "y" but the dataname was "response", from the actual argument. Again, these attributes are mainly useful in printed output, plotting and occasionally in error messages. For example,

```

FUNCTION qqnorm( y/REAL/
    ...
    ylab/CHAR,1,DATANAME(y)/
    ...

```

uses as default value for the argument *ylab* the name of the actual dataset given to *qqnorm* for argument *y*. (If the actual argument is a general expression, it will not have a dataname. In this case, DATANAME(y) points to a null string containing only the EOS character.)

6.2.3. Missing Values.

S data structures can contain missing values (NAs). By default, however, any vector structure appearing in an interface routine will be checked for missing values, and an error generated if any are found. This reflects our feeling that there are no reasonable default strategies for dealing with missing values across the breadth of data analytic techniques. Functions which feel they know how to deal with missing values bear the onus of specifically allowing them. In the declaration of an argument or other structure, this can be done by the specifier, NAOK, indicating that missing values are allowed.

One reasonable way of handling NAs is to pass them on in corresponding values of the result of the function. Consider the function *diff*, for example. It is reasonable to allow missing values in the argument, and to set a value to NA in the result if either of the data values subtracted to get the difference are missing. Two further expressions in the interface language allow this: NA(expr) returns TRUE or FALSE according to whether the argument value is missing, while NASET(item) sets its argument to missing. (NA should *not* be declared logical.) Missing values should not be assigned or otherwise operated upon in the interface routine: they should only be handled by NA or NASET. In the example of *diff*, the description of the argument and the calculation change as follows:

```

FUNCTION diff( x/TS,NAOK/, ...
...
do i=1, LENGTH(x)-lag
    if(NA(x[i]) | NA(x[i+lag]) ) NASET(y[i])
    else y[i] = x[i+lag]-x[i]
...

```

If either $x[i]$ or $x[i+lag]$ is missing, so is $y[i]$; otherwise we compute the difference.

6.2.4. Structures and Components.

S provides for general hierarchical structures, whose components are extracted in the S language by using the "\$" operator. Techniques are available for operating on hierarchical structures in the interface language as well. The type STR in the description list corresponds to a general hierarchical structure. One can then extract the component (if any) of this structure having a specified name: to extract component named x from a structure z :

```
x/FROM(z),.../
```

For example, consider the function *regprt*. This function takes the hierarchical structure produced from a regression and uses various components of the structure to print summaries of the regression. The regression structure is the argument to *regprt*. In the interface routine, it is necessary, for example, to extract the components *coef* and *resid* from the regression structure. This can be done as follows:

```

FUNCTION regprt( z/STR/ )
...
STRUCTURE( coef/REAL,FROM(z)/,
    resid/REAL,FROM(z)/ )
...

```

The argument z is coerced to be a hierarchical structure. Then the structure *coef* is extracted from z by looking for the component called *coef*, and coercing this to mode REAL. If there is no component name matching *coef*, an error is generated. It is possible to make the search optional, just as in the description of arguments to the function. For example,

```
STRUCTURE( int/FROM(z),LGL,1,TRUE/ )
```

looks for *int* as a component of z , but allocates a single logical value (TRUE) if there is no component matching *int*.

The process of extracting components from a structure is very much like that of looking for arguments. It is possible to use exactly the standard argument processing facilities to examine the components of a structure. The chief application of this technique is to functions which can take either a list of arguments or a structure whose components are treated as arguments. Section 6.3.4 will discuss the details of the technique.

6.2.5. Modes Computed at Execution.

(This is a specialized, advanced topic. Skip it unless you need it.) In the discussion and examples so far, the mode of the data in a structure was constant (i.e., known at the time the interface routine was compiled.) Interface routines should be organized so that this is the case, whenever possible. The interface language automatically

generates references to the correct data mode in response to the use of the name of the structure or an expression like $x[i,j]$. If the mode is not known until execution, the writer of the interface must take special steps to ensure that a check of the actual mode is made and that correct run-time code is generated for each case. The interface language does not do this automatically.

There are, however, examples of functions that do allow different modes at execution. One example is the *encode* function. This S function returns a character vector produced by encoding any non-character arguments and then concatenating corresponding data elements in the arguments; e.g.,

```
> encode("Value",1:3)
```

```
"Value 1" "Value 2" "Value 3"
```

The arguments to *encode* can be of any mode, and one obviously wants to encode each argument correctly for each mode. Here is a restricted version of the function, which takes two arguments.

```
FUNCTION encode( x/ANY/, y/ANY/ )
```

```
STATIC( integer i,ii,nmax; POINTER istrng )
```

```
nmax=max0(LENGTH(x),LENGTH(y))
```

```
STRUCTURE( c/CHAR,nmax/ )
```

```
do i=1,nmax {
```

```
    ii = mod(i-1,LENGTH(x))+1
```

```
    SWITCH MODE(x) {
```

```
        CASE REAL: ENCODE( R(x[ii]) )
```

```
        CASE INT: ENCODE( I(x[ii]) )
```

```
        CASE LGL: ENCODE( L(x[ii]) )
```

```
        CASE CHAR: ENCODE( C(TEXT(x[ii])) )
```

```
    }
```

```
    ii = mod(i-1,LENGTH(y))+1
```

```
    SWITCH MODE(y) {
```

```
        CASE REAL: ENCODE( R(y[ii]) )
```

```
        CASE INT: ENCODE( I(y[ii]) )
```

```
        CASE LGL: ENCODE( L(y[ii]) )
```

```
        CASE CHAR: ENCODE( C(TEXT(y[ii])) )
```

```
    }
```

```
    c[i] = istrng(BUFFER,BUFPOS)
```

```
    CLEAR
```

```
}
```

```
RETURN(c)
```

```
END
```

There several new techniques in this example, some of which will be treated in more detail later. The arguments are declared mode ANY (and, by default are vector structures). They will not be coerced to any specific mode, as long as they can be treated as vectors. The interface routine will produce a result, *c*, which is as long as the longest of the arguments.

The *do* loop encodes individual data elements from *x* and *y* and creates a character string pointer to the concatenated result. This pointer becomes the corresponding data value in the vector *c*.

Data values in *x* and *y* are reused cyclically, if the arguments are of different length (this is the purpose of the code to compute *ii*). The encoding of different elements is done by means of algorithm language ENCODE expressions, to be discussed in section 7.2.1. The point to note here is that the computations for different modes are done by means of the SWITCH MODE expression:

```
SWITCH MODE(x) {
```

```
  CASE REAL: #the computations for the case that x is REAL
```

```
  ...
```

The construction is similar to the *switch* statement in RATFOR, but is specially processed by the interface language so that in the code which follows *CASE REAL*; all references to *x* treat data values as real, and similarly for the other modes. This construction is the essential technique for handling modes which vary at execution.

A corresponding situation arises when a data structure is to be allocated with a mode that is not known until execution time. Some such cases can be handled by *LIKE*, but there is also a mechanism to allocate a structure with a mode calculated during execution; namely,

```
STRUCTURE( y/MODECALC(expression), .../)
```

The argument to *MODECALC* can be any computed (integer) expression; typically, it is *MODE(x)*, where *x* is some other dataset. The following example, however, illustrates another situation. In the S function *pmax*, we want to find the maximum data value of all the arguments. Suppose, again, we restrict the function to two arguments. A simple way to write the function would be as follows:

```
FUNCTION max( x/REAL/, y/REAL/ )
```

```
  STATIC( integer i,nmax,ix,iy )
```

```
  nmax=max0(LENGTH(x),LENGTH(y))
```

```
  STRUCTURE(z/REAL,nmax/)
```

```
  do i=1,nmax {
```

```
    ix = mod(i-1,LENGTH(x))+1
```

```
    iy = mod(i-1,LENGTH(y))+1
```

```
    z[i] = amax1( x[ix],y[iy] )
```

```
  }
```

```
  RETURN(z)
```

```
END
```

As with *encode*, the data elements of *x* and *y* are used cyclically. Notice that *max* will return a result of mode REAL even if both its arguments were INT. There is nothing seriously wrong with this and user functions might well be better to keep things simple. However, for large integers, there may be a loss of precision on converting to real values. For this and for aesthetic reasons, one might prefer to return integer values when the arguments were integer. This can be done with *MODECALC*.

The idea is that if both arguments are either logical or integer, the result will be integer, but if one or more arguments are real, so is the result. (To simplify, logical

arguments will be coerced to integer; there is no loss of information.) To do this simply, we take advantage of the fact that the built-in modes in S actually have numerical values (try printing the S built-in datasets REAL, INT, LGL, etc.). The numerical values are chosen to reflect a hierarchy useful in coercing:

LGL < INT < REAL < CHAR

Thus, the mode of the results is just the maximum of the modes of the arguments. Since logicals are being coerced to integers, the mode will be at least as large as INT. Thus, the computed mode is

`mode=max0(INT,MODE(x),MODE(y))`

When *mode* is computed, it implies not only the mode in which the result should be allocated, but also the mode to which *x* and *y* should be coerced. This introduces another new idea: coercing data structures after they have been found as arguments or allocated. This is done by the COERCE statement:

`COERCE( name/type,mode/ )`

where *type* and *mode* can be any of the types and modes introduced previously.

It would be possible to use MODECALC as a mode in the coerce statements. However, we will have to use SWITCHMODE to handle the different modes in any event. It is simpler then to put in a separate COERCE for each of the possibilities, because after seeing the statement

`COERCE(x/REAL/)`

the interface language knows to treat data values in *x* as reals. These ideas lead to the following version of *max*.

`FUNCTION max( x/ANY/, y/ANY/ )`

`STATIC( integer i,nmax,mode,ix,iy )`

`nmax=max0(LENGTH(x),LENGTH(y))`

`mode=max0(INT,MODE(x),MODE(y))`

`STRUCTURE(z/MODECALC(mode),nmax/)`

`SWITCH MODE(z) {`

`CASE REAL: COERCE(x/REAL/); COERCE(y/REAL/)`

`do i=1,nmax {`

`ix=mod(i-1,LENGTH(x))+1`

`iy=mod(i-1,LENGTH(y))+1`

`z[i]=amax1( x[ix],y[iy] )`

`}`

`CASE INT: COERCE(x/INT/); COERCE(y/INT/)`

`do i=1,nmax {`

`ix=mod(i-1,LENGTH(x))+1`

`iy=mod(i-1,LENGTH(y))+1`

`z[i]=max0( x[ix],y[iy] )`

`}`

`}`

`RETURN(z)`

`END`

These techniques are admittedly a little subtle. The basic rule is that one can only refer to data items computationally in the interface language when their mode is known. Thus, if the mode of a structure is not known at compile time, a SWITCH MODE expression must be used, with separate and essentially duplicate code for each mode of interest. Where other structures are involved, it is usually necessary to coerce these to the corresponding modes. The interface language's knowledge of modes is determined strictly by position in the interface routine. After encountering a statement which says that the mode of *x* is REAL, each succeeding statement referring to data values of *x* treats those as REAL. If (horrors!) there were a jump into the middle of this code during execution, when *x* was not actually REAL, chaos would result.

There are a small number of special algorithms accessible to interface routines which in fact treat data structures of arbitrary mode. The technique is to pass these routines the attribute

VALUE(*x*)

which is a pointer to the data values. Along with the mode, this allows the algorithms to treat the data correctly. An example of such an algorithm is

call pcopy(*from,to,mode,length*)

which copies data between two data structures, given the value pointers *from* and *to*. New algorithms of this type can be written as well. This approach may in fact turn out to be a good solution to handling data of arbitrary mode.

6.3. Function Arguments.

6.3.1. Arguments in the FUNCTION Statement.

As we have seen in the previous sections, the list of arguments in the FUNCTION statement is of the general form:

(*name*₁ /description list₁/, ... *name*_k/ ... /)

When the interface routine for the function is invoked, the *actual arguments* (given in the user's call) are scanned to find the argument corresponding each of the *formal arguments* (in the definition of the function) in turn. If an actual argument of the *name=value* form matches *name*_{*i*} exactly, or by an unambiguous partial match (see below), this actual argument is used for the formal argument *name*_{*i*}. Otherwise, the first unnamed argument not previously matched is used. If no argument matches, the attribute MISSING(*name*_{*i*}) is set to TRUE and the default action is taken. If a default structure was defined, this is allocated. If OPTIONAL was specified, but no default structure, the structure is allocated of type NULL (presumably something will be done later to check for the argument being missing). If neither default form was specified, a FATAL error is generated.

The partial match procedure works as follows. The interface routine has a table of the names of all the arguments appearing anywhere in the function definition. To match a specific argument, the actual arguments are scanned to find either an exact match, or an actual argument whose name matches a non-empty leading part of the name of this formal argument (and of *no other* formal argument). For example, suppose the arguments to a function have names "x", "y", "intercept" and "iter". Consider the following actual arguments:


```
i=1
int=1
intercept=1
```

The first fails to match any argument, since its name is a leading part of both "int" and "iter". The second successfully matches "intercept". The third fails to match any argument (a smarter partial match would perhaps help.) The idea of the partial match is to allow users to employ short abbreviations, but permit the formal names of the arguments to be informative.

When the search and coerce are completed for all the arguments in the argument list, the operation ENDARGS is invoked to check that all the actual arguments have been matched to some formal argument (to detect extra arguments or misspelled argument names).

6.3.2. Interrupting and Resuming Argument Processing.

It may be useful to make the argument processing depend on some computed tests; for example, on whether or not some previous argument was supplied. The ENDARGS check may be suppressed by giving "&" as the last argument:

```
FUNCTION abc( x/MATRIX/, wt/REAL,OPTIONAL/, &)
```

This special symbol does not match any actual argument, but simply indicates that the remaining arguments (if any) should be left for further processing. By including "&" as the last argument on the FUNCTION statement, the interface can now do any additional computations. Argument processing is resumed by the statement

```
ARG( name1/ ... /, ..., namek/ ... / )
```

Searching and coercing arguments in the ARG statement are identical to the same process in the FUNCTION statement.

For example, suppose function *abc* takes an optional argument *wt*. If *wt* is present, a further optional argument, *wtmin*, may be supplied.

```
FUNCTION abc( x/MATRIX/, wt/REAL,OPTIONAL/, &)
```

```
  if(!MISSING(wt)) {
    ARG( wtmin/REAL,1,0./, & )
```

```
    ...
  }
ENDARGS
```

Notice that the ARG statement, like the FUNCTION statement, will finish by performing the ENDARGS operation unless ARG also gets a final argument "&". In the example, to check for the end of the argument list when *wt* was not supplied, the interface routine invokes ENDARGS explicitly.

It is possible to suppress positional matching of arguments and require arguments to be given only in the *name=value* form, simply by following the name by "=" in the FUNCTION statement:

```
FUNCTION display( label=/CHAR,OPTIONAL/, ...
```

An actual call of the form

```
display("Hello", ...)
```

would *not* match "Hello" to the formal argument *label*. Arguments specified in the interface routine in the *name=* form are not matched partially: the actual name must be given exactly (this prevents anomalies in the case of arbitrarily many arguments; see below).

6.3.3. Arbitrarily Many Arguments.

S functions like *c*, *print* and *save* take a list of arbitrarily many arguments and perform some computations on each of them. In the interface language, this requires a loop over all actual arguments, rather than a process of matching each actual argument to a different formal argument. The loop is defined by the matching pair of statements:

```
ALLARG( name )
```

```
...
NEXTARG
```

The effect is as follows. ALLARG looks for the next available argument (excluding those matched by the FUNCTION or ARG statements previously executed). If there is no such argument, control breaks out to the statement after the matching NEXTARG statement. Otherwise, *name* specifies an S structure which is made to correspond to the matching actual argument. The statements down to the matching NEXTARG are now executed and control returns to the ALLARG statement.

This argument matching process differs in several ways from the FUNCTION or ARG statement. It is non-destructive: another subsequent ALLARG loop will match the same arguments in the same way. It ignores the names, if any, given to the actual arguments. It does not coerce the arguments to any specific type or mode.

The non-destructive loop is useful in that one often needs to go over the arguments twice, the first time to determine lengths and/or modes, the second time to allocate and process data.

Since ALLARG does not coerce the structure to a type or mode, neither special attributes (NROW, etc.) nor actual data values will be available automatically for the matched argument. The COERCE statement should be used in the loop to coerce the argument.

```
FUNCTION mixem( & )
```

```
...
ALLARG(x)
    COERCE(x/MATRIX,NAOK/ )
```

```
...
NEXTARG
```

The description list in the COERCE statement is the same as in the ARG statement, except that none of the default description of the structure can be supplied.

Two situations arise with respect to argument names in the use of ALLARG. The simplest is that the argument names are irrelevant to this S function, and should not be supplied. In this case, the interface routine should not match arguments in the *name=value* form. The option NOKEY to ALLARG suppresses match of any named arguments:

ALLARG(x,NOKEY)

Named arguments in functions with arbitrarily many arguments are usually special arguments, matched outside the ALLARG loop. The NOKEY option prevents matching them in the loop and therefore allows checking for misspelling of the name, using ENDARGS:

```
FUNCTION display( label=/CHAR,OPTIONAL/, ..., &)
```

```
...
ALLARG(z,NOKEY)
```

```
...
NEXTARG
```

```
ENDARGS #catch misspelled label
```

Notice that here we recognized *label* only in the *name=value* form. This is usually convenient, since the user can supply *label* anywhere in the argument list, with no danger of confusing it with the other, ordinary arguments.

A different situation occurs when the function uses the name given to the actual argument in computations. For example, the *save* function uses any argument names as the name for the corresponding dataset on the save database. In this case, one allows named arguments and picks up the name, if it is not NULL, in the ALLARG loop:

```
FUNCTION save( pos=/INT,1,2/, &)
```

```
STATIC( POINTER pname )
```

```
ALLARG(x)
```

```
    if(NAME(x)!=NULL) pname=NAME(x)
```

```
    else pname=DATANAME(x)
```

```
...
NEXTARG
```

This form is suitable if the S function wants to take a name from the dataset, but allow the user the option to override. (In the example, the overriding name could not be *pos*.)

6.3.4. Treating Structures Like Argument Lists.

The list of arguments to a function forms a hierarchical structure, with the individual arguments being the components of the structure. Sometimes a similar process to argument matching is useful for the components of other structures as well. For example, a structure produced as output from one function may later be given to another function as an argument. In this case, the structure is *one* argument; all the results of the previous function need to be extracted from this structure. (Contrast the use of CHAIN (section 6.4.2), which passes individual results from the first function as individual arguments to the second.)

One way to extract components from a structure is the FROM expression in a STRUCTURE statement:

```
STRUCTURE( Dim/FROM(x),INT,OPTIONAL/ )
```

This looks in the structure *x* for a component whose name matches "Dim". The component, if found, is coerced according to the description list, just as it would have

been in a FUNCTION or ARG expression. As the example shows, OPTIONAL and default descriptions may be used as well. The attribute MISSING will be set according to whether the component is found.

For example, the function *bxp* takes as an argument the single structure *z*, which always has components *stats*, *conf* and *n*, and which may contain other components, such as *names*, as well. One way to implement this argument situation would be:

```
FUNCTION bxp(z/STR/, &)

STRUCTURE( stats/FROM(z),MATRIX/
            conf/FROM(z),REAL/
            n/FROM(z),INT/
            names/FROM(z),CHAR,OPTIONAL/
            )
```

If we need to pick up the same component from more than one source, FROM can be given a second argument, which is the name to look for in the structure:

```
STRUCTURE( adim/FROM(a,Dim),INT/
            bdim/FROM(b,Dim),INT/ )
```

This statement generates new structures *adim* and *bdim* corresponding to the component named "Dim" in *a* and *b* respectively. The repeated use of FROM may be verbose if many arguments are to be extracted from one structure. The expression

```
ARGSTR(z)
```

shifts the entire argument-searching process to the components of *z*, rather than the original argument list. Subsequent ARG statements will always refer to the structure named in the most recent ARGSTR statement. In particular,

```
ARGSTR
```

returns argument searching to the original argument list.

There are several differences in the effects of FROM and ARGSTR. The argument search mechanism in the ARG statement ensures that each actual argument is matched only once, by setting the name field of the actual argument to the special value USED, so that it will not match anything on subsequent searches. In contrast, the FROM expression just searches for the required component.

After ARGSTR, names will be matched by the partial match strategy against the list of *all* argument names (not just those which were mentioned as possible components of *z*). The FROM expression does no partial matching; component names must match exactly.

FROM is simpler for most applications; ARGSTR is useful when the structure is really standing in for a (potentially fairly complex) argument list as was the case with *bxp* above.

6.4. Function Results and Related Statements.

6.4.1. The RETURN Statement.

The general form of the RETURN statement is one of:

```
RETURN( comp1, ..., compk ) #or
RETURN( comp1, ..., compk, & )
```

The effect of the statement is to add the structures *comp_i* as components of the data structure returned by the function. If "&" is included, execution of the interface routine then continues. Otherwise, the function returns to the S executive.

The components *comp_i* can be of the forms:

```
structurei
namei = structurei
= structurei
```

where *structure_i* is the name of an existing S data structure in the interface routine, and *name_i*, if present, is any legal name. In all cases, *structure_i* is made a component of the result of the function. If no name is specified, the component name is *structure_i* itself; in the second and third forms the given *name_i*, or an empty name is used as the component name. It is irrelevant whether *name_i* is the name of an existing structure or not; it is only used for naming the component of the result generated.

As an example of the usefulness of renaming arguments, suppose that *x* is an argument to a function, but that we wish to return a structure with a component called "x", containing different values from those in *x*; say, values in a vector *plotx*. This can be done, without re-allocating and copying, by:

```
RETURN(x=plotx, ... )
```

Other applications arise in functions that CHAIN to other functions (6.4.2).

In addition to the general forms above, RETURN takes two special arguments:

```
FILTER
PAR
```

The special component PAR returns graphical parameter settings (see section 6.5.2). The component FILTER returns as results all *unmatched* arguments to the S function. It goes along with "&" in the FUNCTION statement and the CHAIN statement to pass along unused arguments as results.

Some details of the effect of RETURN may be relevant. Structures are associated with components of the result through the dynamic structure pointed to by *structure_i* at the execution of the RETURN statement. The data values are not copied to a temporary structure; rather, the pointers to the structure are copied into the structure to be used as the result of the function. This is usually irrelevant in practice, except for the implication that it is unwise to change the contents or to reallocate a structure appearing a previous RETURN statement.

Only dynamically allocated structures may be returned, not scalars or arrays appearing in STATIC declarations.

If only one component is returned, this component becomes the entire result of the function, as opposed to returning a structure with a single component. Again, this is not usually an important distinction in practice.

6.4.2. CHAIN: Invoking Another Function.

In a number of circumstances, a new S function is an extension of, or wants to use an existing function. Often, this can be expressed in the sense that the new function would like to finish by invoking the existing function. For example, the probability-plotting functions *qqplot* and *qqnorm* prepare two sets of co-ordinates for a scatter plot and chain to the scatter plot function *plot* to do the graphics. In the interface language, this is done by the CHAIN statement:

```
CHAIN(name) #or
CHAIN(name, structure1, ..., structurek)
```

The first argument, *name*, is the name of an S function. The remaining arguments, if any, are interpreted exactly as arguments to the RETURN statement. The effect of CHAIN is identical to that of RETURN, with the additional side effect that on return to the S executive, the function *name* will then be called. The components of the result of the current function will become the arguments to function *name*.

Notice that there is a difference between CHAIN-ing to a function and calling the function with the result of the previous function. For example, suppose the interface routine for function *abc* has the statement:

```
CHAIN(def,x,y,wt)
```

The effect is to call function *def* with three arguments, named *x*, *y* and *wt*. If, on the other hand, *abc* had the statement

```
RETURN(x,y,wt)
```

and was used in an S expression of the form:

```
def(abc( ... ))
```

then *def* is called with one unnamed argument, with three components. Either may be desirable, but the effect is different. See the INSERT statement (6.4.3) for simulating the second form.

The use of CHAIN is clearly one reason for being able to rename the results of an S function (6.4.1). The names in the CHAIN statement may be chosen to be those required by the function to be invoked next, or may be empty to generate positional arguments.

6.4.3. INSERT: Building Structures.

Corresponding to the ability to extract components from a structure (section 6.3.4), it is possible to insert structures as components of other structures.

```
INSERT(structure, component1, ... )
```

The syntax of *component_i* is the same as for the RETURN statement. Each structure specified in the component arguments is inserted into *structure* as a component. As with the RETURN statement, the component will have the name specified in the *name=component* form if this form is used. The only difference is that null names may not be used with INSERT. The INSERT statement is equivalent to a RETURN statement except that the returned structure is taken to be the first argument to INSERT, not the value of the function as a whole.

6.5. Graphics Functions.

S functions to do plotting are slightly different from other functions. They will communicate, through graphical algorithms (section 8), with the device driver functions. They frequently will take graphic parameters as optional arguments. Also, they may recognize a special argument structure which is useful for specifying plotting positions.

6.5.1. Declaring a Graphics Function.

When a graphics function is first declared, the FUNCTION utility must be given the special option "-g":

```
S FUNCTION -g myplot myplot.i ...
```

declares that *myplot* will call graphical algorithms.

6.5.2. Graphical Parameters.

In general, users of graphical functions should be able to supply settings for graphical parameters (line type, color, plotting character, etc.) as optional arguments. The strategy recommended is that the parameter changes should be made before plotting is done, and restored (i.e., reset to the values which existed before the graphical function was called) after all plotting is complete. This does not apply to parameters which should stay in force to allow additional plotting on the current figure (e.g., axis parameters or user co-ordinate range). See *par* in the detailed documentation for the parameters involved.

To pick up any graphics parameters in the arguments to a function, include the special argument PAR; for example,

```
FUNCTION myplot( x/REAL/
                y/REAL/
                PAR
                )
```

This does not look for a single argument in the usual way. Instead, it invokes an algorithm which scans the argument list for any of the graphical parameters, specified by name, extracts the corresponding values from the arguments and specifies these values for the corresponding graphical parameters (see also section 8.3.1). At the same time, the *previous* values of all these parameters are saved internally in the interface routine.

Upon exit from the graphics function, one of two actions will be taken. If the special result structure PAR is returned, the graphics parameters will retain their new settings, and the previous values will be passed back as a special component of the result. This is used exclusively with the CHAIN statement. If *myplot* chains to some other graphics function, for example *axes*, which also accepts graphical parameters, then the second function will leave the new parameter settings in effect during plotting and then restore the original values when it, in turn, exits. (This all happens automatically, so long as each of the graphics functions in the chain includes PAR as an argument.) The code is then of the form:

```
FUNCTION myplot( x/REAL/, y/REAL/, PAR, & )
```

```
... #plotting stuff
```

```
CHAIN(axes, PAR, FILTER)
```

```
END
```

Notice that we allowed extra arguments to *myplot* through "&", to be used by *axes* (such as *asmain* or *xlab*).

In the case that PAR is not returned, the original graphics parameters are automatically restored on exit from *myplot*. In any case, the parameters will be restored if an error or interrupt occurs during execution of the plotting function.

6.5.3. Plotting Data Structure.

For many applications, a special *plotting data structure* is useful. This is a hierarchical structure with two components, named "x" and "y", corresponding to the x- and y-coordinates for a set of plotting positions. Generally, graphics functions which expect to get a set of plotting positions try to match any of the following possibilities:

- a plotting structure as above;
- arguments *x* and *y* giving the two sets of co-ordinates;
- one argument to be interpreted as a time-series (either a true one, or a vector with "start" time 1).

To look for all of these possibilities automatically, use the statement PLOTARGS in place of the corresponding argument specifications. In *myplot* above, to allow for all these possibilities rather than just two vectors, the interface routine would be written:

```
FUNCTION myplot( & )
STRUCTURE(x,y)
PLOTARGS
ARG( PAR, & )
... #and so on, as before
```

After PLOTARGS executes, vector structures *x* and *y* will be available for the x- and y-coordinate values respectively. The interface routine can go on to set up the plot (section 6.5.4) or can produce graphical output immediately, if the plotting is to be added to the existing plot.

By default, missing values are not allowed in PLOTARGS. If the application could reasonably have missing values the interface routine should say:

```
PLOTARGS( NAOK )
```

6.5.4. High Level Graphics Functions: SETUP and LOGPLOT.

For graphics functions which want to produce a new plot, there are two interface language statements that carry out automatically typical setup requirements. The statement

```
SETUP
```

coming after structures *x* and *y* have been defined (typically by PLOTARGS), will choose user co-ordinates and axis parameters to fit these values. In addition, SETUP will automatically look for arguments of the form *xlim*=... or *ylim*=..., and take these

as overriding the axis limits implied by the data to be plotted. Thus,

```
FUNCTION myplot( & )
STRUCTURE(x,y)
PLOTARGS
SETUP
ARG( PAR, & )
... etc.
```

The interface language statement

LOGPLOT

declares and sets two static logical scalars, *logx* and *logy*, such that *logx* will be TRUE if the user specified an argument to the S function so as to specify log transformation of the x-axis. For example if the user's call included an argument of the form *log="x"* or *log="xy"*, then *logx* will be TRUE. Similarly, *logy* will be TRUE if log transformation of the y-axis was specified.

Writing and Using Algorithms

7. Writing and Using Algorithms

Section 6 described the implementation of new S functions under the assumption that the subprograms (algorithms) were already available. In practice, however, one usually needs to write or at least adapt algorithms for the underlying computational problem at the same time. This section presents techniques for writing algorithms and a brief description of the algorithms supplied with S. While algorithms could be written in any form such that the resulting subprogram could be called from FORTRAN, the S software environment provides many special features and tools to make the writing easier. Collectively, we refer to this as the S environment, and to the language used for writing algorithms (based, for example, on RATFOR) as the *algorithm language*.

The algorithm language and the algorithm library described in this section are *not* restricted to programming new S functions. Many of the facilities are helpful in writing stand-alone programs and utilities outside of S. Instructions for running such stand-alone programs will be given in section 7.1.2.

7.1. The Algorithm Language: Basics.

7.1.1. Languages.

The language in which most of the algorithms supporting S are written is an enrichment of the RATFOR structured preprocessor for FORTRAN. The enrichment consists of some general features (described in this section) and a number of specialized facilities (section 7.2). It is also possible to use the S environment through the EFL preprocessor for FORTRAN or as an enrichment of the C language. Most of the discussion will be of the RATFOR form.

Files containing programs or subprograms developed for in algorithm language are identified as to base language by the suffix of the file names. Files ending in ".r" are compiled using RATFOR; those ending in ".e" are compiled with EFL and those ending in ".c" are compiled using C. The general and special facilities described below are all available either with RATFOR or EFL. However, the facilities available with C are more limited (on the other hand, C provides some compensating facilities within the language). Section 7.1.6 describes the available C features.

7.1.2. MAKE: Generating S Functions and Stand-Alone Programs.

Generally, it is not necessary to compile and load subprograms individually in the S environment. Instead, one associates algorithms (i.e., files containing subprograms in some form of the algorithm language) with either an S function or a stand-alone program via the FUNCTION or PROGRAM utilities. The utility MAKE then generates an up-to-date version of the S function or program. An S function is declared by

S FUNCTION name file1 file2 ...

where *name* is the name of the S function and *file1*, etc. are files of source code in the interface language or the algorithm language. For example if S function *myreg* has an

interface routine on file *myreg.i* and RATFOR-based supporting algorithms on files *newreg.r* and *comp2.r*, then the S function is declared (once) by

```
S FUNCTION myreg myreg.i newreg.r comp2.r
```

(See section 6.1.) The similar facility for stand-alone programs is

```
S PROGRAM name file1 file2 ...
```

where *name* is now the name you want for the executable program. For example, suppose we want a stand-alone program test called *testreg* which tests *newreg.r* and *comp2.r*, and that the main program for the test is on *testreg.r*:

```
S PROGRAM testreg testreg.r newreg.r comp2.r
```

Once the FUNCTION or PROGRAM declaration has been completed for the S function or the stand-alone program, the actual program can be generated by

```
S MAKE name
```

with *name* either the S function or the executable program name. The MAKE command is a specialization and extension of the UNIX command *make*. It takes such compiling and loading steps as needed to produce the executable program. Note that the files on which *name* depends are not supplied to MAKE; this information has been stored away by FUNCTION or PROGRAM. Whenever one of the files is updated, repeating the MAKE command will cause the executable version to be updated.

The files on which *name* depends should include all local algorithms needed to execute the program. They should not include any S algorithms (described in this manual); the program libraries including these algorithms are searched automatically by MAKE.

If the set of files on which *name* depends changes, the information used by MAKE must be updated by rerunning FUNCTION or PROGRAM with its revised list of files. Note that FUNCTION or PROGRAM should not be redone when the *content* of files change, only when the list of files changes.

7.1.3. Error Handling.

Fatal errors, warnings and messages to the user are handled by statements similar to those in the interface language:

```
FATAL(message)
WARNING(message)
MESSAGE(message)
```

What happens after the FATAL message is printed, however, is different. If the algorithm was called from an S function (i.e., directly or indirectly from an interface routine), the message from the algorithm will be followed by a message

```
Error in name
```

where *name* is the name of the S function. In the case of a stand-alone program, the FATAL message is followed by an abort (exit) from the program, with no further message. More complex error messages may use the ABORT statement, which prints all its arguments on the error file (see section 7.2.1) and then acts like a fatal error.

7.1.4. Symbolic Constants; Declarations.

The algorithm language provides a number of constants in symbolic form. These increase readability. More crucially, since most of the constants are potentially machine-dependent, programming with them in symbolic form increases portability. The more frequently used constants, and their meaning, are:

PRECISION # the smallest pos. real such that $1. + \text{PRECISION} > 1$

BIG # the largest real

SMALL # the smallest positive real

BIGEXP # the largest exponent (base 10)

LARGEINT # the largest decimal integer

INT, REAL, CHAR # modes

NCPW # the number of characters per (FORTRAN) word

NBPC # the number of bits per character

NDIGITS # the number of decimal digits to represent reals

EOS # the string terminator character

ESCAPE # the character to escape special characters

PI # 3.14159...

DEG2RD # $\text{PI}/180$.

In addition to these constants, there are two special declarations:

CHARACTER(name,width,dim1,dim2,...)

declares *name* to be a variable or an array to hold character data. (Because different FORTRAN implementations may have different formats for this, the CHARACTER statement aids portability.) The *width* gives the maximum number of characters in an element of *name*. The arguments *dim1*, etc., if given, are the dimensions of the array of character mode elements.

POINTER p1,p2,...

declares variables or arrays *p1*, etc. to be pointers for use with dynamically allocated data (section 7.2.4). POINTER variables are actually integers to FORTRAN, but we recommend using the declaration to document the use of the variables as pointers.

7.1.5. Debugging.

When an algorithm is called from an S function, the DEBUG statement may be used, as in the interface language (section 6.1.7). The interactive debugging facility will allow the user to query the contents of any of the S data structures in the interface routine. In order to use this feature, the interface routine must contain some form of DEBUG statement. If no actual debugging is desired in the interface routine, the statement

DEBUG(ON)

can be used in the interface routine.

For stand-alone programs or for data which does not correspond to S structures, the printing facilities of section 7.2.1 may be used. There is no (portable) way to look at a symbol table for non-S structures, but local interactive debugging facilities may be helpful (e.g., the UNIX program *adb*).

7.1.6. C Language Facilities.

Stand-alone programs may be written in the C language and C-based subprograms may be used as support for S functions, provided the writer handles the interface between the C routine and the FORTRAN-based routines calling it. See, for example, the *Fortran 77* documentation in the UNIX manuals (volume 2).

The algorithm language facilities available to C-based routines are principally the symbolic constants of section 7.1.4 and the error handling routines of 7.1.3. The statements FATAL, WARNING and MESSAGE are similar to their use in RATFOR or EFL, but slightly more general. The first argument is taken as a format string. It is possible to include additional arguments, which will be interpreted as data items to be printed using the format string. For example:

```
if( maxstep < 0)
    FATAL(Value of maxstep is %ld - must be >=0, maxstep)
```

(Avoid embedded commas in the string.) The example illustrates that FATAL, WARNING and MESSAGE are complete C language statements (no terminating semi-colon is added).

The special algorithm-language facilities described in section 7.2 are not available to C-based routines, unless specifically noted.

7.2. Special Facilities.

The algorithm language has a number of specialized facilities which are not made available automatically to all routines. When one or more of these facilities is needed, the programmer can make it available by putting the statement

```
INCLUDE(name)
```

among the declarations of the subprogram. The *name* specifies the particular facilities wanted, as described in the subsections below; e.g., *print* for the printing facilities. The INCLUDE statement causes a set of specialized algorithm language statements to be made available and may also insert the declarations for some global variables (i.e., common blocks) at this point in the subprogram.

More than one facility can be included in a single program, by multiple arguments to INCLUDE, separated by commas, or by multiple INCLUDE statements.

7.2.1. Printing and Encoding.

The algorithm language provides a facility for formatted printing and encoding of data. This replaces and extends FORTRAN-style formatted output. To have access to the printing facilities

```
INCLUDE(print)
```

should appear among the declarations of the program. No FORTRAN language output statements should appear in algorithms used with S functions.

7.2.1.1. Basic Message Printing.

The simplest and most common use of the facilities is to print some message, including data values, either as an error message or as part of the user's standard printed output. Standard printed output is produced by:

```
PRINT(arg1, arg2, ... )
```

where *arg1*, etc. are items to be printed. Similarly, printed output for error messages is produced by

```
EPRINT(arg1, arg2, ... )
```

The distinction is that error output is directed to the terminal even if the standard output has been redirected to a file. EPRINT is used for errors and other messages to the user; PRINT is for the printed display of results from the analysis. (See below for output from S functions.) Formatted output can be combined with the generation of a FATAL error by the ABORT statement:

```
ABORT(arg1, arg2, ... )
```

acts like EPRINT, followed by an error exit equivalent to that produced by FATAL. It is also possible to use ABORT with no arguments to exit after printing messages with EPRINT.

The arguments *arg1*, etc. are either quoted strings or format items (data values and optional formatting parameters). data values are encoded according to their mode, either in free format or by a user-specified format. Stick to the simpler free-format items listed here until you need to produce particularly elegant print-out:

```
"This is a message" #string
R(x) #real value
I(jj) # integer value
L(x>0) # logical value
C(msg) #characters with EOS
C(buff,length) #chars with count
Q(msg) #characters, printed in quotes
Q(buff,length)
```

Strings may contain any printing character except ",", "#" or unbalanced parentheses (see 7.2.1.4). The argument to R, I or L can be any expression of the appropriate mode. The printing facility examines the data value and chooses a suitable format. A leading space will be inserted to separate successive items, except character literals enclosed in quotes. For example:

```
PRINT(I(nrow)," by",I(ncol)," matrix")
```

It is possible to direct printed output to a file other than standard-output or standard-error, by using

```
FPRINT(ifile, arg1, arg2, ... )
```

instead of PRINT or EPRINT. Before FPRINT is used, the desired output file must have been opened. The integer *ifile* is the file descriptor returned by the opening routine *sopen* or *sattac* (see section 7.2.3).

Output for S functions. There are two kinds of output from algorithms to be used with S functions: error messages and displays. In general it is not a good idea to write new functions which produce printed displays unless there is really something new about the printing itself. Otherwise it is much cleaner to work with (possibly CHAIN to) one of the existing S functions for printing. Informative error messages, on the other hand, can often benefit from printing out the data values involved in the error. These can use EPRINT or ABORT just as above:


```
ABORT("Iteration failed after",I(niter)," steps, still",R(delta)," apart.")
```

If you really need to write a new display function, the output should be directed to the file OUTFC, rather than STDOUT, since S can then redirect the output through the *sink* function. This file is opened automatically in all S functions, but printed output from an algorithm needs to include the definition by the statement

```
INCLUDE(io)
```

among the declarations.

7.2.1.2. Encoding.

The same facilities which produce the printed output can also be used to encode strings. The encoded text can then be used for labelling of graphics, as part of an S structure or for any other purpose. The statement

```
ENCODE(arg1, arg2, ... )
```

encodes data items in the same way as the PRINT, etc. statements, but leaves the resulting text in a character variable, BUFFER. The integer variables BUFPOS and BUFLen give the current and maximum length of the text in BUFFER. Multiple encode statements will concatenate the successive strings in BUFFER. The contents of the buffer may be printed and BUFPOS reset to 0 at any time by use of PRINT or EPRINT without arguments. For example

```
ENCODE("Statistics:")
do i=1,5
    ENCODE(R(stats(i)))
PRINT
```

This illustrates one convenience of the ENCODE statement: building up a line of output iteratively. Normally, one will need to check that the buffer has not become too full.

```
PRINT("High:")
do i=1,nhigh {
    if(BUFPOS>LINEWIDTH-FIELDWIDTH) PRINT
    ENCODE(R(x(i)))
}
if(BUFPOS>0)PRINT
```

The last line prints any fields on the final line of output. Symbolic constants FIELDWIDTH and LINEWIDTH specify the maximum length of an encoded numeric field, and the longest line of printed output.

The other main application of encoding is to make character string data items. The usual tool for this purpose is the supplied algorithm:

```
istrng(text,length)
```

which returns a POINTER value to a stack copy of *length* characters of *text*. For example,

```
ENCODE("Group",I(ngroup))
ip=istrng(BUFFER,BUFPOS)
```

generates the encode text in BUFFER and then assigns to *ip* a pointer to a copy of the

same text, on the dynamic stack. Note that the text in BUFFER does not have a string terminator at the end, unless it is provided explicitly; e.g.,

```
ENCODE("Group",I(ngroup),"EOS")
```

After "Group" and the integer field are encoded, the special character EOS is inserted (EOS is a symbolic constant, not three separate characters).

If encoding is not followed by a PRINT or similar statement, it is essential that the buffer be cleared (otherwise the encoded string appears at the front of the next PRINT message):

```
ENCODE("Group", I(ngroup))
ip=istrng(BUFFER,BUFPOS)
CLEAR
```

is the socially responsible way to do the preceding example.

7.2.1.3. Detailed Format Control.

When detailed control of the printed output is desired, spacing and specific format information can be given. The following arguments to PRINT, ENCODE, etc. control spacing:

```
SP(n) #space forward n characters
T(m) #tab to position m
TM(iw) #tab to a multiple of iw
```

The TM argument is used to print output in columns *iw* wide. The T operation tabs to column *m*, or inserts 1 blank, if the current BUFPOS is greater than *m*. As an example of spacing, consider the "high" values in the example above as an indented display.

```
ENCODE("High:")
do i=1,nhigh {
    if(BUFPOS>LINEWIDTH-FIELDWIDTH) {
        PRINT
        ENCODE(T(7))
    }
    ENCODE(R(x(i)))
}
if(BUFPOS>7)PRINT; else CLEAR
```

Each line after the first starts in position 7; the test for printing the last line is modified correspondingly.

Data formats may be given explicitly in the R, I and L fields. The I and L fields take an optional width argument:

```
ENCODE( I(ij,5))
```

encodes an integer field (right-justified) 5 characters wide. REAL fields take three parameters: width, number of positions after the decimal point and a third argument of E_FORMAT or F_FORMAT to select the format type. Typically, the real format is chosen to give the same format for a set of real numbers. In this case, the format parameters should be chosen by an algorithm, *rdtfmt*. The programmer need not worry about them personally. For example,


```
call rdtfmt(x,nhigh,i1,i2,i3) #compute format for x
ENCODE("High")
do i=1,nhigh {
```

```
    ...
    ENCODE( R(x(i),i1,i2,i3) ) #use format
```

modifies the previous example to use a constant format for all values. The algorithm *rdtfmt* takes a vector of real values, and its length, and returns the three format parameters.

One further specialization of the formats is to give a length of 0. This is equivalent to encoding in free format, except that the leading space which separated items is suppressed. Suppression of internal spaces is a good idea, for example, if the resulting string becomes the name of a dataset or component:

```
ENCODE("Gr",I(ngroup,0))
ip=istrng(BUFFER,BUFPOS)
CLEAR
```

makes character strings of the form "Gr5" or "Gr1066" for corresponding values of *ngroup*.

7.2.1.4. Problems with Encoding.

Because the printing and encoding facilities are implemented by a macro processor, certain problems occur. For example, it is difficult to print a comma or sharp sign. For these purposes, special format items

```
COMMA
SHARP
```

are provided. It is *not* possible to print a comma or sharp in the middle of a literal string (enclosed in quotes).

Another difficult task is printing parentheses. In this case, one solution is to declare character variables that contain the troublesome characters, and to use the "C" encoding directive.

```
CHARACTER(lpar,1); CHARACTER(rpar,1)
...
data lpar,rpar/"(",")"/
...
ENCODE( C(lpar,1), ...
```

7.2.2. Reading and Decoding.

Facilities are provided for input of data items and the breaking up of text into fields corresponding to data values. These correspond in a symmetric way to the printing and encoding facilities above. To give a program access to the reading facility:

```
INCLUDE(read)
```

This activates definitions of the reading and decoding operations and also provides access to a buffer for reading, INBUF, along with its current and maximum length, INPOS and INLEN. The reading and printing buffers are separate, so that reading and printing can be mixed within a single subprogram.

7.2.2.1. Basic Reading of Data Items.

The fundamental viewpoint of the reading facility is that input comes as a stream of characters. Fields within the stream are separated by "white space" (blanks, tabs or new lines). To read a number of fields from the standard input stream (typically the user's terminal):

```
READ( arg1, arg2, ... )
```

A line will be read from the input file. The fields described by *arg1*, etc. will be decoded from the line in the mode requested. If the end of the line is reached, another line will be read, and so forth.

The fields are described similarly to printed fields:

```
R(x) # real
```

```
I(ii) # integer
```

```
L(flag) # logical
```

```
C(buffer,maxlen) #characters
```

```
S(buffer,maxlen) #same with EOS
```

The arguments to the fields must be variables or array elements, since the decoded values will be stored in them. The two character fields differ only in whether a string terminator character is inserted at the end of the field. Notice that the argument is a character buffer (not a pointer) and that a maximum length should be supplied to protect against overflowing the buffer. Other than this, no lengths or format control should appear in the decoding statements. See the section 3.2.4 and the detailed documentation for *extract* in S for a utility to extract data from a file, given column formatting information.

A typical use of the READ facility might be to read user input in response to a prompt, for example:

```
MESSAGE(Enter desired number of iterations & start (x,y):)
```

```
READ( I(niter), R(x), R(y) )
```

The user can supply input on one line or several; the READ statement will not complete until all three fields have been decoded. Errors in interpreting fields produce a FATAL error message. See 7.2.2.2 for line-oriented input and end-of-file.

It is not necessary for the entire input to be read in one statement. The statement

```
DECODE(arg1, arg2, ... )
```

will decode additional fields, starting just after the last decoded field. If there are no more fields on the current line, the next line of input will be read. For example, to read first the number of items and then all the items, allowing as many of the items as desired to be on any line:

```
READ( I(n) )
```

```
for(i=1; i<=n; i=i+1) DECODE( R(x(i)) )
```

READ does not need any arguments: when called without arguments, it fills the input buffer, ready for any DECODE statements. Note that using a second READ statement after the first will throw away any unused fields in the current line. For example, suppose we want to read character fields from the input, but skip the rest of a line after a field which starts with "#".


```

repeat {
    DECODE( C(field,maxlen) )
    if(jgetch(field,1) == "#" ){
        READ; next }
...

```

7.2.2.2. Line Input; End-of-File.

Fields can also be decoded from input which is read a line at a time. This form is useful if a specific number of fields are to be taken from each line. To get a line of text into the input buffer,

```
GETLINE(ifile)
```

where *ifile*, if supplied, is the integer file descriptor for a previously attached file (section 7.2.3). If *ifile* is omitted, input is from the standard input (normally the terminal).

The following example reads pairs of real numbers from each line, until end-of-file is reached.

```

for( i=1; i<=imax; i=i+1) {
    GETLINE(ifile)
    DECODE( R(x(i)),R(y(i)) )
    if(EOF)break
}

```

If GETLINE encounters an end-of-file condition, it sets the internal EOF flag, which should be checked by the calling program, as above. Note that DECODE following GETLINE will not read further lines. When the last field from the current line has been decoded, further DECODE requests will set EOF and immediately return. In particular, no automatic error abort will occur; it is the calling program's responsibility to check EOF.

Decoded fields can be used to generate strings, similarly to the use of encoded fields in section 7.2.1. A decoded field is put into a character buffer called FIELD. The length of the current field is FIELDPOS. Therefore, to read an integer field and make a string on the stack containing it:

```

DECODE( I(ival) )
ip=istrng(FIELD,FIELDPOS)

```

This leaves the integer value in *ival* and a pointer to the string in *ip*.

It is also possible to decode strings which were not originally read from a file; for example, strings which were arguments to an S function. The statement

```
GETSTRING( text )
```

moves the string *text* into the decoding buffer. Subsequent DECODE statements then operate as after the GETLINE statement.

7.2.3. File Access; Standard Files.

The S environment provides the ability to attach and detach files, for example, in the printing and reading operations of the previous sections. Files can be opened in an algorithm or interface routine by

```
ifile=sopen(name,perms)
```

where *name* is the name of the file (a character string terminated by the end-of-string character) and *perms* is READ, WRITE, or READWRITE for read-only, write-only or read-write permission. The function *sopen* (remember to declare it integer) returns a positive integer *file descriptor* if the open was successful and a negative integer if the open failed (e.g., the file could not be opened or permission was refused). When the file input/output is complete, the file is closed by

```
call sclose(ifile)
```

In S interface routines or algorithms called by them, a more automatic procedure for attaching files can be used. This puts the files on a list of open files, which can be automatically detached if appropriate. To use these facilities,

```
INCLUDE(attach)
```

must appear in the interface routine or algorithm. To attach a file

```
ifile=sattac(name,perms,type)
```

where *name* and *perms* are as before and *type* specifies when the file is to be automatically detached: values of AUTO, ONERROR and PERM specify that the file disappears automatically at the end of the expression, on encountering an error and never, respectively. In any case, the file can also be detached explicitly by

```
call sdetac(ifile)
```

Generally, it is bad strategy to attach and detach files in an S function unless there are special requirements. Many troubles will be avoided if, when possible, files are converted to data structures in S before the individual S function sees them (using *read* or *?extract*, for example).

In any case, try to ensure that attached files are faithfully detached, either through the AUTO feature in an S function or by explicitly using *sdetac*.

7.2.4. Dynamic Storage.

The S environment allows dynamic allocation and release of blocks of storage. This is the preferred way to provide working space to numerical and other algorithms. To use the dynamic allocation facility

```
INCLUDE(stack)
```

should appear in the subprogram. Storage is allocated by the function *jstkg*. This returns a *pointer* to dynamically allocated storage sufficient to hold a specified number of data items of specified mode; e.g.,

```
ip=jstkg(n,REAL)
```

returns a pointer to space for *n* items of mode REAL. All variables used as pointers (and functions returning pointers) to the stack should be declared POINTER:

```
POINTER ip,jstkg
```

When storage has been allocated on the stack, elements of the allocated storage can be referenced in the appropriate mode:


```

rs(ip) #REAL
is(ip) #INTEGER
ls(ip) #LOGICAL

```

For example, to allocate $2*n$ integers and initialize to 0, ..., $2n-1$

```

iptr=jstkg(2*n,INT)
for(i=0; i<2*n; i=i+1) is(iptr+i)=i

```

When the dynamic storage is no longer needed, the LAST k blocks of storage allocated may be released by

```

call jstkrl(k)

```

Note that storage is allocated in a *first-in-last-out* form. It is not possible to release a dynamically allocated block unless you also release all blocks allocated later than the one in question.

7.2.5. Data Structures for S.

This section is very advanced, and is intended for those writing S support algorithms which require knowledge of S data structures. It assumes familiarity with these structures, as discussed in section 5.3.

Algorithms (not written in the interface language) which need to refer to the structure of S data should have

```

INCLUDE(struct, stack)

```

in the routine.

Whether in an algorithm or in an interface routine, suppose that *ptr* is a pointer to an S data structure. In an algorithm, *ptr* will be passed in through the argument list or will be the result of previous computations with S structures. All S data structures will have the attributes

```

NAME(ptr)
MODE(ptr)
LENGTH(ptr)

```

for the (pointer to) the name of the structure, the (integer) mode of the structure and the (integer) length (number of data elements). Every structure also has the pointer attribute

```

VALUE(ptr)

```

which points to the values (on the stack of dynamically allocated data) of the data structure.

If *ptr* points to a vector, then the mode will have one of the values REAL, INT, LGL or CHAR. In the first three cases, the data values for the vector are found on the corresponding stack. If

```

n=LENGTH(ptr); v=VALUE(ptr)

```

then the data items are

```

rs(v), ..., rs(v+n-1) # reals
is(v), ..., is(v+n-1) # integers
ls(v), ..., ls(v+n-1) # logicals

```

If the mode is CHAR, then *v* points to a vector of *n* pointers, each of which points in turn to the actual text of the corresponding string. For example, to print each of the character strings using the printing facility of section 7.2.1:

```

for(i=0; i<n; i=i+1)
    PRINT( Q(TEXT(is(v+i))) )

```

Note that the *integer* stack is used to refer to an array of pointers. Similarly, to allocate an array of pointers, use INT as the mode argument to *jstkgf*.

If the original structure was not a vector, but a hierarchical structure, then MODE(ptr) is STR. In this case, VALUE points to a *directory* whose entries are the components of the structure. Expressions in the algorithmic language allow reference to the entries: suppose that *dirptr*=VALUE(ptr) where ptr is a structure. Attributes defined for directories include:

```

FIRSTENT(dirptr) # pointer to the first component
LASTENT(dirptr) # pointer to the last component
ENTRY(dirptr,j) # pointer to the j-th component

```

A typical operation on a hierarchical structure is to loop over all the components, performing some computations on the component data structures. One way to do this is

```

dirptr=VALUE(ptr)
for( i=1; i<=LENGTH(dirptr); i=i+1) {
    compnt=ENTRY(dirptr,i)
    ## now do the work on compnt
}

```

Note that *compnt* is a pointer to the substructure, so that all the attributes (NAME(compnt), MODE(compnt), etc.) are defined.

The loop over the components shown above allows computations over one level of substructures. It is possible to loop over all the components of a structure at any level. This process is usually called a depth-first search or tree traversal. This process is less often needed than simply looping over one level of components, and requires some knowledge of tree structures and of the *m4* macro system (see the UNIX documentation).

S provides for a depth-first scan of any hierarchical structure. To use the scan

```

INCLUDE(tree)

```

should appear in the program. The scan moves over all components of the structure. As each component is encountered, its mode is tested. If the mode is STR (the component is a structure itself) the scan then moves DOWN to examine all subcomponents. Otherwise the scan moves ALONG to the next component at the current level. Finally, when the scan has gone over all the components at the current level, it moves UP to the next component at the level above.

The algorithm language statement

TREEWALK(ptr)

where *ptr* is the pointer to an S data structure, carries out the scan. The individual routine defines up to three *actions*; namely, DOWN, ALONG and UP. Each of these is specified by defining a macro of the given name, containing the statements to be executed. The definitions must appear before the TREEWALK statement.

The attribute CURRENT is the pointer to the structure component currently being examined. Other attributes defined are LEVEL, the current depth within the structure, the pointers, PARENT and NEXT, to the structure above CURRENT and to the (directory of) the structure to which the treewalk will go down, and MAXLEV, the maximum depth of structure allowed.

A very simple example prints the level and the name of all components:

```
subroutine trpr(ptr)
  POINTER ptr

  INCLUDE(tree,print)
  define('DOWN','EPRINT( I(LEVEL), C(TEXT(NAME(CURRENT)))) ')
  define('ALONG','EPRINT( I(LEVEL), C(TEXT(NAME(CURRENT)))) ')

  TREEWALK(ptr)
  EPRINT("End of structure")
  return
end
```

7.3. Available Algorithms.

This section presents brief discussions of algorithms which may be of general interest. Omitted from this discussion are most of the algorithms which are close analogues of individual S functions. Look at the source code for the S function to see what algorithms are used, if you want an algorithm for a similar purpose. The source code for algorithms will be found in directories *psl* and *s/basic*. See *System* in section 9a.

7.3.1. Data Handling; Character Data.

Arrays of all modes may be filled with constant values or copied into other arrays.

```
call rfill(data,x,length) #REAL data
call ifill(idata,ix,length) #INTEGER data
call lfill(ldata,lx,length) #LOGICAL data
```

These all fill arrays with the single data item given. To copy data,

```
call rcopy(xfrom,xto,length)
call icopy(ifrom,ito,length)
call lcopy(lfrom,lto,length)
```

handle REAL, INTEGER and LOGICAL vectors.

Character data. The utilities to fill and copy character data are:

```
call cfill(char,buffer,length)
call concat(to,ito,from,ifrom,length)
```

Copying character data is handled somewhat more generally, since one may wish to specify the starting character position in one or both strings. The routine *concat* copies *length* characters from the *ifrom*-th character of *from* to *to*, starting at the *ito*-th character. The character positions are numbered from 1.

Two other basic operations for moving character data are provided:

```
ich=jgetch(string,i)
```

gets the *i*-th character from *string*. Note that both *ich* and *jgetch* should be declared

```
CHARACTER(ich,1); CHARACTER(jgetch,1)
```

The opposite process of putting a character into a string is

```
call jputch(string,i,ich)
```

There are a number of algorithms and macros to convert among various forms of character data and pointers to strings.

```
TSTRING(anything)
```

puts an EOS on the end of some literal text and encloses it in quotes. To convert text to a pointer to a string on the stack

```
ptr = istrng(text, length)
```

If *length* is given as -1 , the text will be scanned for a terminating EOS character. If character data is initially packed into a buffer in the form of *n* fields, each *length* characters long, the data may be converted into a vector of *n* pointers to character strings on the stack:

```
ptr = ch2vec( buffer, n, length)
```

This returns a pointer to *n* pointers on the integer stack, the *i*th pointer pointing to a character string made from the *i*th field in *buffer*. Trailing blanks are removed from each field, and an EOS character appended.

To test the equality of two strings, use the logical function *streq* (remember to declare it):

```
if(streq(text1,text2)){ ... matched ... }
```

See also the partial match techniques used in S (section 6.3.1).

7.3.2. Sort and Order.

There are algorithms for sorting data or producing the ordering permutation. These are analogues of the S functions *sort* and *order*, with the crucial difference that in the FORTRAN-based algorithm language, there must be different versions of the routines for different modes. For sorting,

```
call sort(x,length) #REAL data
call sorti(ix,length) #INTEGER data
```

sort vectors of REAL or INTEGER data. There is then a general sort

```
call sortf(ix,length,fun)
```


In this routine, *fun* is the name of an external function. This function will be called by the sort routine to compare two data items from the vector to be sorted. The function *fun* takes two integer or POINTER arguments and returns -1, 0 or 1 according as the first argument precedes, is equal to or follows the second argument. The usefulness of doing the compares in a function is that any ordering can be followed, not just the obvious numerical one.

The common example is that of comparing and thus sorting character strings. The algorithm *chcomp(ip1,ip2)* compares two strings, given two pointers *ip1* and *ip2*. This leads to the following use of *sortf* to sort a vector of (pointers to) character strings.

```
subroutine sortc(ptr,length)
  POINTER ptr; integer length
```

```
  external chcomp
```

```
  call sortf(ptr,length,chcomp)
  return
end
```

Note that data of mode POINTER is actually of FORTRAN mode integer.

Parallel to the sorting routines are routines which return, in addition to the sorted data, a vector of integers giving the ordering permutation (see the S function *order*).

```
call orderr(x,length,iorder)
call orderi(ix,length,iorder)
call orderf(ix,length,iorder,fun)
```

In each case, *iorder* is an integer vector of *length* items, returned filled with the permutation which orders the vector *x* or *ix*.

(S structures only.) Two routines exist to sort and order data of arbitrary mode, given the VALUE pointer:

```
call sortv(ptr)
call orderv(ptr,iorder)
```

The argument *ptr* is a pointer to an S vector of mode REAL, INT or CHAR.

7.3.3. Range of Data.

For plotting purposes, one needs to compute the range of data values.

```
call rangev(x,length,xmin,xmax)
```

returns in *xmin*, *xmax* the minimum and maximum value of the vector *x*. Given several sets of data, one can obtain the cumulative minimum and maximum of all the sets by the function *rangev*. For example, suppose *x1*, *x2* and *x3* are three vectors.

```
call rangev(x1,n1,xmin,xmax)
call rangev(x2,n2,xmin,xmax)
call rangev(x3,n3,xmin,xmax)
```

The value returned from *rangev* for *xmin* will be the minimum of the data values in the argument and the input value for *xmin* and similarly for *xmax*.

If the data may have NAs included, the range of all the actual values may be obtained from

```
call narang(x,n,xmin,xmax)
```

7.3.4. Probabilities; Quantiles; Pseudorandom Numbers.

7.3.4.1. Available Algorithms.

For the most part, the routines to compute these quantities can be inferred from the S functions. As explained in section 2.3.1 and 3.3.1, the distributions are coded and functions to produce probabilities, quantiles and random values have names formed from "p", "q" and "r" prepended to the code. The underlying algorithms have similar names, but produce only a single value as result.

Not all of the distributions are represented by explicit algorithms; random values from the remaining distributions are computed by the inverse probability law; i.e., *qDIST(runif(...))* is equivalent to *rDIST(...)*. Random number generators are provided explicitly for

```
chis
expos # exponential
gamm # gamma
norms # one algorithm for normal
normk # alternative algm for normal
pois # poisson
stab # stable family
```

The calling sequence for each of these includes all parameters *except* for location and scale parameters, followed by an integer seed argument.

```
xchis=rchis(idf,iseed)
```

returns a Chi-squared random value with *idf* degrees of freedom.

7.3.4.2. New Pseudorandom Generators for S.

This is an advanced topic. For the most part, new facilities for generating pseudorandom numbers in S can and should be done by generating uniform or other random numbers and transforming these according to the generating scheme desired. Typically, this is a good application for an S macro.

If a new function is to be written to generate numbers, it must mesh with the technique used by S to make all random numbers flow from one consistent sequence of values (and, in particular, to avoid having the S function start its generator at the same place every time). The basic idea is to use a special dataset, named "Random.seed", to keep the seed values for the generators. This dataset is read in at the beginning of each function generating random values, and written out at the end.

An S function that generates random numbers should be declared with the "-r" option of the FUNCTION utility (see the detailed documentation). Before computing any pseudorandom values, the interface routine should include the statement:

```
GETSEED
```

After computing all desired pseudorandom values, the interface routine should include:

PUTSEED

All new random generators should proceed by using the existing basic (uniform, etc.) generators. See *uni* in the algorithm detailed documentation.

7.3.5. Matrices and Arrays.

Numerous algorithms are included for numerical computations on matrices. There are also two routines for in-place transposition of matrices.

```
call transr(x,nrow,ncol)
call transi(ix,nrow,ncol)
```

where *x* and *ix* are real and integer matrices with *nrow* rows and *ncol* columns.

The following numerical routines are of general use. Most of them require some knowledge of numerical linear algebra, to apply the algorithms in new areas. We give only the names here; see the algorithm documentation for argument lists.

```
backsl; backsu # back-solve triangular system
chol # choleski decomposition
condu # estimate condition number
eigenl # eigenvalues
dot, dotv, dotwv # inner products
gs # q-r decomposition
gsls, gslsi, gslsw gslsiw # least-squares
matp, matpt # matrix product
svd # singular-value decomposition.
```

There are other routines associated with individual S functions, either with corresponding names or mentioned in the S function documentation.

Graphical Algorithms

8. Graphical Algorithms.

This section discusses the creation of graphical algorithms, either for use within S functions, or in a stand-alone graphics environment. In both cases, graphical algorithms are based upon a set of portable, device-independent graphical subroutines designed for use in data analysis. The algorithms built from these subroutines are device-independent, and can communicate with many different graphical devices through device driver programs.

The underlying graphical subroutines range from general, high-level routines to specific, low-level routines. For example, some subroutines produce entire plots, complete with axes and titling. At the other extreme are subroutines that draw a line, plot a text string, etc. Intermediate level routines are available for tasks such as setting up an axis or plotting a set of titles. This structure provides building blocks for constructing new graphical algorithms.

A large set of graphical parameters allows control over special characteristics of the resulting plots (color, character size, etc.). At the same time, the graphical subroutines provide reasonable default values for parameters that the algorithm writer does not wish to specify. In this case, the algorithm *user* (either the user of an S function incorporating the algorithm or the author of a subroutine calling the algorithm) retains the ability to change these parameters. Strategy parameters control more general aspects of the plotting, such as the overall shape of the plot (e.g., square, maximum sized for the device), the layout of multiple figures on a single page, and methods used to generate pretty axis labels. The current value of any parameter can be specified or queried by graphical algorithms.

All plotting is done in convenient coordinate systems: the major portion of the plot is carried out in the user's own coordinate system, titling and axis labelling in a different, specially designed system.

Graphical algorithms are written in the algorithm language described in section 7, and thus are easy to read and write. In addition, facilities of the algorithm language provide special convenience and efficiency in the use of graphical parameters.

Section 8.1 describes the basic concepts of the graphical subroutines; 8.2 describes the construction of a graphical algorithm, using an example; 8.3 extends the example to both the S and stand-alone environments; 8.4 treats more advanced details of the graphical subroutines and parameters. A number of subroutines are described in section 8.5. Section 8.6 discusses stand-alone graphical algorithms, and section 8.7 is concerned with device drivers.

8.1. Basic Concepts of Graphical Algorithms.

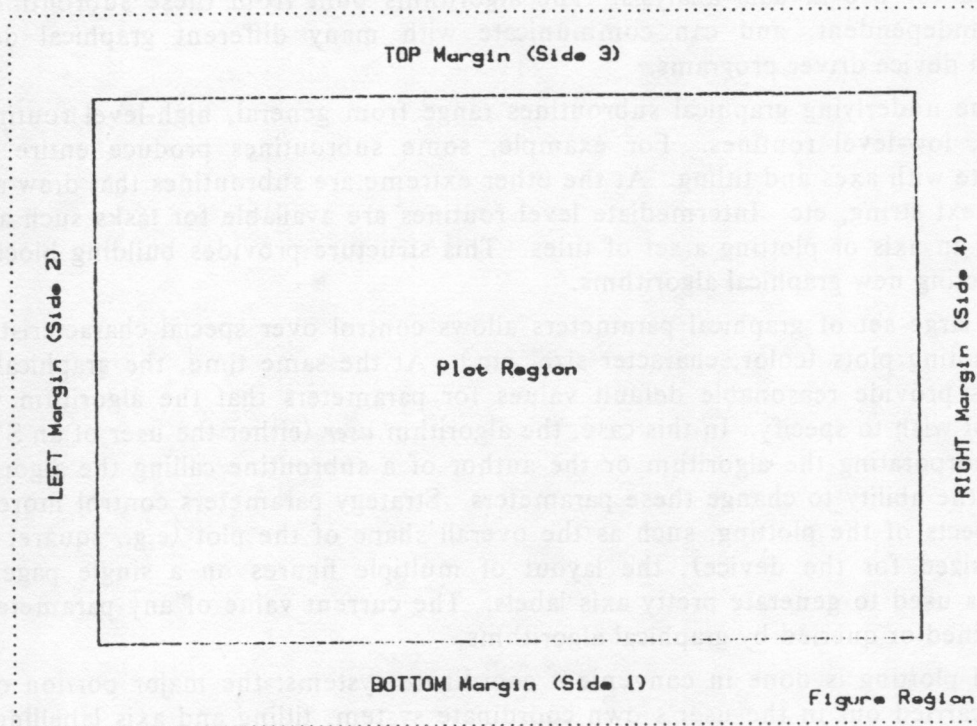
There are three basic concepts provided by the S graphics facility:

- the way in which graphical output is structured into *figures* containing labelling information which surrounds a *plot* of data-analytic information.
- the relevant *coordinate systems* in which graphical operations are expressed.
- the use of *graphical parameters* to control the graphical output.

Familiarity with these simple concepts provides the basis for understanding how to work with and write graphical algorithms.

8.1.1. Figures and Plots.

A typical plot consists of a central portion containing the primary graphical information (a scatter plot, curves, a histogram), surrounded on four sides by auxiliary information: titles, axis scaling, legends, etc. We will refer to the central region as the *plot*, and the border surrounding it as the *margins*. Together, the plot and margins make up a complete *figure*. This is illustrated below.

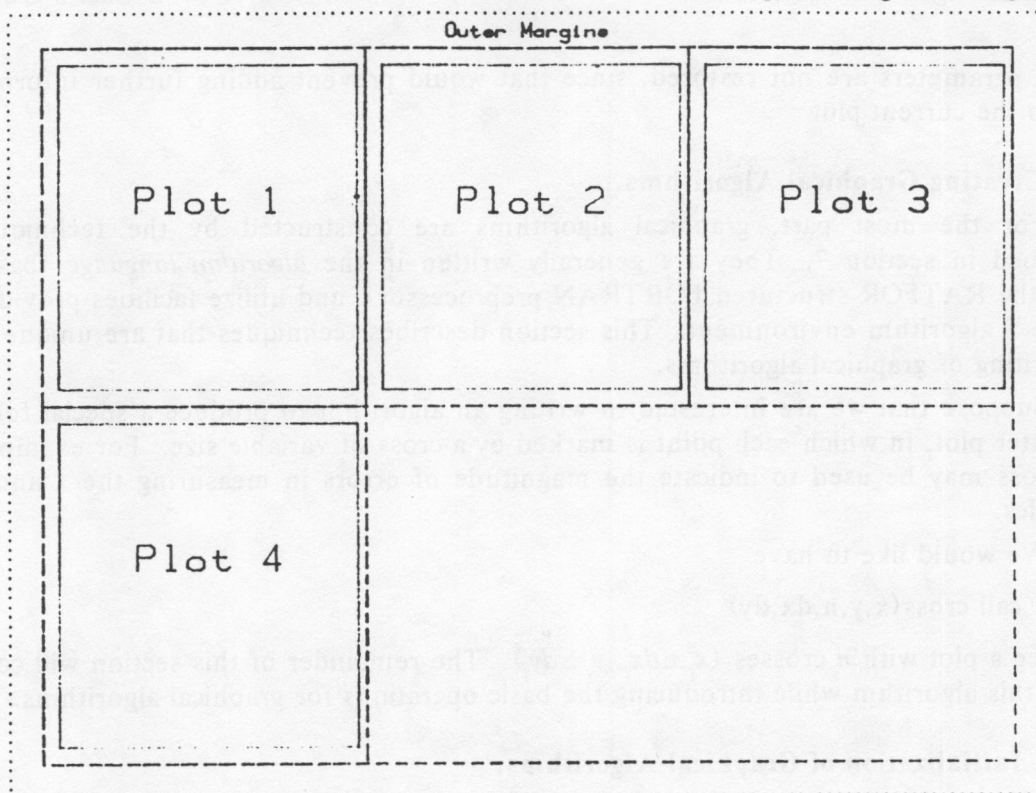


The figure is a complete piece of graphical output, and occupies all or part of a page of output or the surface of a graphical device. A device may be organized to display one or more figures. The *outer margins* border the edge of the page, and provide an area for overall titling when more than one figure is present. The region inside the outer margins may be divided up among one or more figures (see the example below).

8.1.2. User and Margin Coordinate Systems.

The natural coordinate system for the plot region and its information is in the units of the user's data. For example, a map may be plotted in degrees of latitude and longitude, a plot of height vs. weight in inches and pounds, etc. These coordinates are known as *user* coordinates.

The titling information, however, is often more conveniently expressed by its relationship to the plotting region. Thus, a title may be centered two lines above the plot. The *margin* coordinates are given by the side (bottom, left, top, or right), the distance from the corresponding edge of the plot, and a position along the side. Since the information plotted in the margins is most often text, it is convenient to describe the distance out from the edge of the plot in lines of text. The position along the side of the plot can be described in several ways, either as a fraction of the way along the side, or by the corresponding user coordinate. Margin coordinates are also used in the outer margins.



There are other coordinate systems that are occasionally useful. It is possible to relate the user coordinate system to the physical dimensions of the device, e.g., to ensure desired spacing or size for graphical output. To explicitly position a figure on the page, it is useful to specify the positioning in fractions of the way up or across the page. It is also possible (though unusual) to give the plot position explicitly as a fraction of the figure.

8.1.3. Graphical Parameters.

Perhaps the most important concept in the construction of graphical algorithms is that of *graphical parameters*. The precise effect of a graphical operation is not defined until a number of decisions have been made by the graphical system. For example, in plotting a scatter of points, the graphical system must decide upon the character (or symbol) to plot at each point, what size the character should be, its rotation, color, etc.

The graphical subroutines use a set of graphical parameters to determine, at a given time, exactly how to carry out specific operations. Values of the parameters may be queried or specified by a graphical algorithm, or may be left at their current values. Reasonable default values exist initially for all parameters. Parameters *not* modified by an algorithm may still be specified by a higher-level routine or by the user of an S function.

In order for the concept of allowing outside control of graphical parameters to work, graphical algorithms must follow a strict discipline whenever changing the parameter values: the parameter value is queried and saved, it is changed, the operation using the changed value is carried out, and the saved value of the parameter is then restored. Restoration of the old parameter value prevents the parameter setting from having a permanent effect. The only parameters changed without restoration are those relating to coordinate systems and axes, specified by high-level algorithms.

These parameters are not restored, since that would prevent adding further information to the current plot.

8.2. Creating Graphical Algorithms.

For the most part, graphical algorithms are constructed by the techniques described in section 7. They are generally written in the *algorithm language* (based upon the RATFOR structured FORTRAN preprocessor), and utilize facilities provided by the S algorithm environment. This section describes techniques that are unique to the writing of graphical algorithms.

Suppose that we are interested in writing an algorithm to produce a special form of scatter plot, in which each point is marked by a cross of variable size. For example, the cross may be used to indicate the magnitude of errors in measuring the x and y variables.

We would like to have

```
call cross(x,y,n,dx,dy)
```

produce a plot with n crosses $(x_i \pm dx_i, y_i \pm dy_i)$. The remainder of this section will construct this algorithm while introducing the basic operations for graphical algorithms.

8.2.1. Initialization of Graphical Algorithms.

The statement

```
INCLUDE(graphics)
```

should appear in the declaration section of the algorithm (along with type statements, common declarations, etc.). This enables the graphical parameter facility, which is later accessed by the two operations QUERY and SPECIFY, described in section 8.2.5.

Algorithms like *cross* that produce a new plot, must also initialize the graphical system at execution time by means of

```
call beginz
```

This routine has two effects: it interacts with the device driver to receive the current values of the graphical parameters, and it causes an advance to a clear figure on the device. If no device driver is currently in effect, *beginz* will cause a fatal error. If no plotting has yet been done on the device, *beginz* will establish all parameters at their default values.

Thus far, our routine looks like:

```
#cross scatter plot with crosses at x +/- dx, y +/- dy
subroutine cross(x,y,n,dx,dy)
integer n
real x(n),y(n),dx(n),dy(n)
```

```
INCLUDE(graphics)
```

```
call beginz
```

8.2.2. Setting-up Coordinate Systems and Axes.

Once the graphical subroutines have been initialized, it is necessary to set up a coordinate system appropriate to the plot. For this, we need first compute the minimum and maximum values to appear on each axis. In many cases the algorithms *rangee*, *rangev*, or *narang* (section 7.3.3) can be used to find the range of data to be plotted. However, for this plot we need to find the extremes of the crosses, and it would be unduly conservative to assume that the largest cross goes with the most extreme data points. Thus, for the x-axis of *cross*:

```
xmax = x(1); xmin = x(1)
for(i = 2; i <= n; i = i + 1){
    xmin = amin1(xmin, x(i) - dx(i))
    xmax = amax1(xmax, x(i) + dx(i))
}
```

Now, given *xmin* and *xmax*, the statement

```
AXIS(BOTTOM,xmin,xmax)
```

will set up a coordinate system that includes at least the values from *xmin* to *xmax*. Also, graphical parameters are stored to define the "pretty" values at which the axis should be labelled. The axis is not drawn yet; this is normally done after the plot is finished (section 8.2.4).

A similar computation is then done for the y-axis

```
... compute ymin, ymax ...
AXIS(LEFT,ymin,ymax)
```

(See section 8.4.4 for information about setting up logarithmic or time axes, or for details of how labels are chosen.)

8.2.3. Drawing the Picture.

With the coordinate system set up appropriately, we can now begin the actual plotting of the crosses. The calls

```
for(i = 1; i <= n; i = i + 1){
    call segmtz(x(i) + dx(i), y(i), x(i) - dx(i), y(i), 1)
    call segmtz(x(i), y(i) + dy(i), x(i), y(i) - dy(i), 1)
}
```

draw the sets of horizontal and vertical line segments used in our plot.

As in this example, it is often straightforward to draw the desired picture once coordinate axes have been set up. The basic subroutines for this are:

```
call pointz(x,y,n)
call linesz(x,y,n)
call segmtz(x1,y1,x2,y2,n)
call arowsz(x1,y1,x2,y2,n)
```

which plot a set of *n* points, draw connected line segments through a set of points, or draw a set of *n* non-connected line segments (or arrows) from the points (*x1(i)*,*y1(i)*) to the points (*x2(i)*,*y2(i)*). All of the x- and y-coordinates are expressed in the previously established user coordinate system. In addition, the routine


```
call crclsz(x,y,r,n)
```

draws a set of circles.

It is frequently useful to place text on the plot. The routine

```
call textz(x,y,string)
```

plots a single character string *string* at the user coordinate position (x,y). The string is centered by default. *String* should contain an end-of-string (EOS) character, as described in sections 6.2.2 and 7.3.1.

Even the most complicated plots are normally produced by suitable combinations of these basic point, line, arrow, circle, and text plotting routines. Section 8.5 describes a number of routines that produce higher level graphical objects through the use of the basic routines described in this section.

8.2.4. Titles and Axis Labels.

With the body of the plot completed, it is only necessary to provide suitable labels. First, the coordinate axes (set up in section 8.2.2) are labelled by

```
call saxisz(side,ticks,labels)
```

The argument *side* takes on the values BOTTOM, LEFT, TOP, or RIGHT. Arguments *ticks* and *labels* are logical values (TRUE or FALSE) specifying whether the tick marks and/or the numeric labels for the axis should be drawn.

For the cross plot,

```
call saxisz(BOTTOM,TRUE,TRUE)
call saxisz(TOP,TRUE,FALSE) # only ticks on top
call saxisz(LEFT,TRUE,TRUE)
call saxisz(RIGHT,TRUE,FALSE) # only ticks on right
```

Titles can be placed in the margins surrounding the plot by

```
call mtextz(side,line,string)
```

Once again, *side* is BOTTOM, LEFT, TOP, or RIGHT. Real-valued argument *line* tells the number of lines of text to leave between the edge of the plot and the title. Together, *side* and *line* give a margin coordinate. *String* is a character string (with end-of-string character) that is centered and plotted parallel to the specified side of the plot.

Thus, to place a descriptive label on our plot:

```
call mtextz(BOTTOM,4.,TSTRING(Cross Represents Error for Each Observation))
call boxz
```

TSTRING creates a terminated string, and *boxz* produces a box surrounding the plot.

8.2.5. Specifying and Querying Graphical Parameters.

Suppose now that we would like character-string identifiers at each point on our cross plot. The identifiers should extend from the center of the cross up and to the right at a 45 degree angle. The calling sequence for the routine changes to

```

subroutine cross(x,y,n,dx,dy,id)
integer n
real x(n),y(n),dx(n),dy(n)
POINTER id(n)

```

(See section 6.2.2 for a discussion of vectors of pointers to character strings.)

The basic subroutine *textz* can be used to plot the strings. However, we must alter several graphical parameters to make the text appear as intended. First, string rotation must be set to 45 degrees, rather than the default 0 degrees (horizontal). Second, the string should be left-justified, not centered. This can be accomplished by

```
SPECIFY( srt(45), adj(0) )
```

to specify the graphical parameters *srt* (String RoTation) and *adj* (string ADJustment). Rotations are measured in degrees counter-clockwise from horizontal. Adjustment varies from 0. (left-justified), through 0.5 (centered), to 1. (right-justified).

If we left-justify the strings at the center of the cross, we will find that the characters overwrite the cross. We would like to move out from the center by some multiple of the character size. To do this, query the current character size in user-coordinates:

```
QUERY( cxy(cx,cy) )
```

Now the identifiers can be placed on the plot by

```

SPECIFY( srt(45), adj(0) )
QUERY( cxy(cx,cy) )
for(i=1; i<=n; i=i+1)
    call textz(x(i)+cx,y(i)+cy,TEXT(id(i)))

```

One remaining problem is that once the string rotation and adjustment parameters have been changed from their default values, any successive text plotting will no longer have the default values of parameters *srt* and *adj*. As mentioned above, the paradigm is to remember the old values of parameters that are to be changed, to set the new values, and finally to restore the parameters to their old values.

```

QUERY( srt(oldsrt), adj(oldadj) )
SPECIFY( srt(45), adj(0) )
... text plotting here ...
SPECIFY( srt(oldsrt), adj(oldadj) ) # restore parameters

```

Parameters are generally named by 3-letter mnemonic strings. As in the example, their values are specified or queried by

```

SPECIFY( parm(values), parm(values), ... )
QUERY( parm(variables), parm(variables), ... )

```

Here *parm* is the mnemonic parameter name, *values* is a list of values that the parameters should take on, and *variables* is a list of variables that are to be set with the current values of the parameters.

There are a number of parameters available for dealing with text besides *srt* and *adj*. Character rotation within a string is controlled by parameter *crt*. By default, this parameter is equal to the most recently specified value of *srt*; thus it is important to specify *crt* after *srt* to give different values to the two parameters.

The size of characters is generally also adjustable, and is controlled through the parameters *cex* (Character EXpansion relative to standard size), *csi* (Character Size in Inches), and *csr* (Character Size Relative to the size of the figure region). Parameters *cex* and *csr* are normally recommended, since *csi* depends more on the size of the output device. For example, to make characters which are double the standard size

```
SPECIFY( cex(2) )
```

QUERY can be used to make characters double their *current* size:

```
QUERY( cex(cursiz) )
SPECIFY( cex(2*cursiz) )
```

With all of the character-oriented parameters, the precise results depend on the graphics device used at execution time. Certain devices have only discrete sets of allowable character sizes and rotations. In general, the device drivers will attempt to make the display obey the graphical parameters, but they will not take extraordinary measures to do so. (Characters are not drawn line-by-line in order to make their size and rotation continuously variable).

There are a number of other parameters that affect the basic graphical subroutines of section 8.2.3. For line drawing, *lty* takes on integer values (dependent on the particular device) to specify various dotted and dashed Line TYpes. Parameter *lwd* controls Line WiDth.

Both lines and text are affected by the *col* parameter, which, on devices with the capability, causes color changes.

The full set of graphical parameters and their default values is presented in section 8.4.

8.2.6. Wrapping Up.

The last graphical call in a high-level algorithm should be

```
call finisz
```

This signals the end of the algorithm, and causes the device driver to note any final changes to parameter values.

With this call, we now have our algorithm

```

#cross scatter plot with crosses at  $x \pm dx$ ,  $y \pm dy$ 
subroutine cross(x,y,n,dx,dy,id)
integer n; real x(n),y(n),dx(n),dy(n)
POINTER id(n)

INCLUDE(graphics)

call beginz

xmin = x(1); xmax = x(1)
for(i = 2; i <= n; i = i + 1){
    xmin = amin1(xmin,x(i) - dx(i))
    xmax = amax1(xmax,x(i) + dx(i))
}
AXIS(BOTTOM,xmin,xmax)

ymin = y(1); ymax = y(1)
for(i = 2; i <= n; i = i + 1){
    ymin = amin1(ymin,y(i) - dy(i))
    ymax = amax1(ymax,y(i) + dy(i))
}
AXIS(LEFT,ymin,ymax)

for(i = 1; i <= n; i = i + 1){
    call segmtz(x(i) + dx(i), y(i), x(i) - dx(i), y(i), 1)
    call segmtz(x(i), y(i) + dy(i), x(i), y(i) - dy(i), 1)
}

QUERY( srt(oldsrt), adj(oldadj), cxy(cx,cy) )
SPECIFY( srt(45), adj(0) )
for(i = 1; i <= n; i = i + 1)
    call textz(x(i) + cx,y(i) + cy,TEXT(id(i)))
SPECIFY( srt(oldsrt), adj(oldadj) )

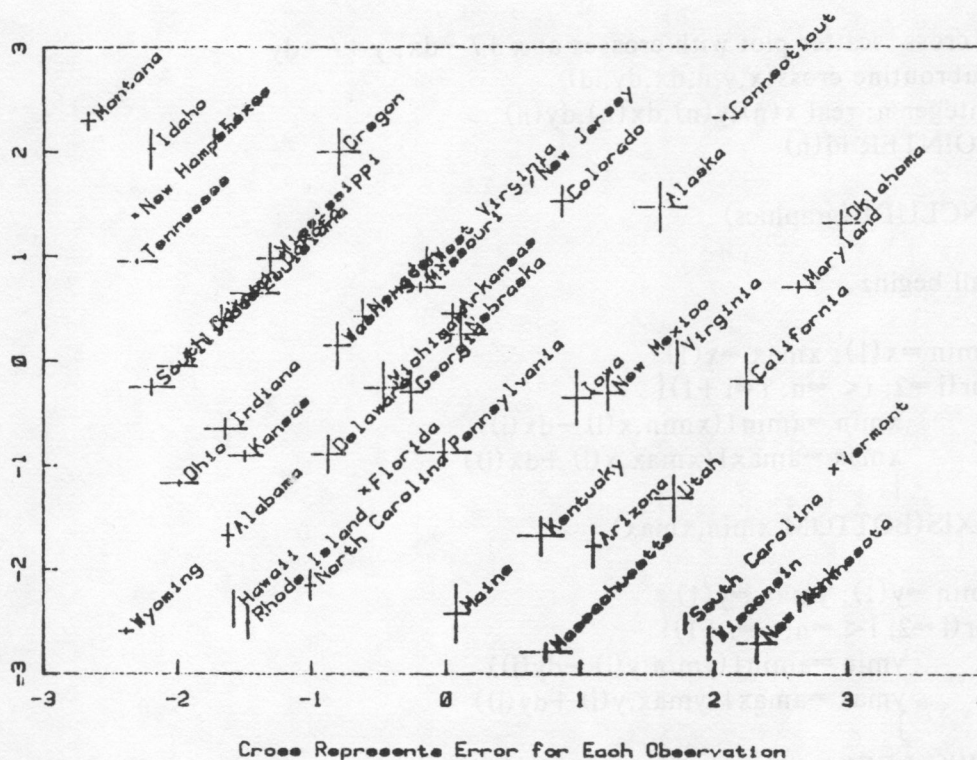
call saxisz(BOTTOM,TRUE,TRUE)
call saxisz(TOP,TRUE,FALSE)
call saxisz(LEFT,TRUE,TRUE)
call saxisz(RIGHT,TRUE,FALSE)

call mtextz(BOTTOM,4.,TSTRING(Cross Represents Error for Each Observation))
call boxz

call finisz
return
end

```

With some artificial data, it produces the following plot.



8.3. The Structure of a Graphical Algorithm.

In section 8.2, we constructed a graphical algorithm *cross* to produce a new form of high-level graphical figure. In this section, we will put it to use.

As it was presented in section 8.2, *cross* is appropriate as a stand-alone high-level algorithm. By creating a main program to call *cross*, we can produce a plot like that shown above.

```
# main program to call cross algorithm
real x(50),y(50),dx(50),dy(50)
POINTER id(50)
... set up x, y, dx, dy, and id vectors ...
... by reading data file, etc. ...
call cross(x,y,50,dx,dy,id)
stop
end
```

We may wish, however, to use the *cross* algorithm in an S function, or to adapt the algorithm to augment an existing plot. The following discussion considers how these changes can be accomplished.

A technical detail: in order to access the graphical subroutines, it is necessary to use the flag "-g" in the *PROGRAM* or *FUNCTION* utilities. To create, respectively, a stand-alone program *cross* or an S function *cross*, type

```
S PROGRAM -g cross main.r cross.r
S FUNCTION -g cross cross.i cross.r
```

(see section 7.1.2).

8.3.1. High-Level Graphical Algorithms; S Functions.

In order to turn an algorithm into an S function, it is necessary to write an interface routine for it. As described in section 6.5, the S interface language provides a number of facilities for the construction of graphical algorithms. Specifically, the *PLOTARGS* and *SETUP* statements provide a convenient mechanism for recognizing a variety of arguments to graphical functions and for setting up axes. *PLOTARGS* allows (x,y) data to be recognized either as two separate arguments or as a single graphical data structure (section 3.4.4).

```

FUNCTION cross(&)
STRUCTURE(x,y)
PLOTARGS # gets (x,y) data
ARG(
    dx    /REAL/
    dy    /REAL/
    id    /CHAR/
    PAR
    &     # allows main =, xlab =, etc.
)

```

The additional argument *PAR* allows user-specified parameter values to be given as arguments to this function, and the final argument "&" allows other (unused) arguments.

By default, the *SETUP* statement automatically uses the range of arguments *x* and *y* to set up limits for the axes. (It also calls *beginz* to signal the start of a high-level graphical routine.) We would like, instead, to use a computation that allows room for the crosses. To accomplish this, we can use the optional arguments

```

SETUP(xmin,xmax,ymin,ymax)

```

At this point, it becomes obvious that a subroutine to compute our axis limits is a good idea, since it could be used either by the stand-alone algorithm or by the S interface routine.

```

call crange(x,dx,LENGTH(x),xmin,xmax)
call crange(y,dy,LENGTH(y),ymin,ymax)
SETUP(xmin,xmax,ymin,ymax)

```

where *crange* is the routine

```

#crange compute range of x +/- dx
subroutine crange(x,dx,n,xmin,xmax)
integer n; real x(n),dx(n),xmin,xmax

xmin=x(1); xmax=x(1)
for(i=2; i<=n; i=i+1){
    xmin=amin1(xmin,x(i)-dx(i))
    xmax=amax1(xmax,x(i)+dx(i))
}

return
end

```

Had we constructed the *crange* subroutine earlier, it could have simplified our stand-alone algorithm.

Now that the coordinate system is set up, we want to plot the crosses and identifiers. A subroutine constructed out of the inner layer of *cross* is useful for this purpose.

```
#cross2  produce cross plot (assumes coordinate system set up)
```

```
subroutine cross2(x,y,n,dx,dy,id)
```

```
integer n; real x(n),y(n),dx(n),dy(n)
```

```
POINTER id(n)
```

```
INCLUDE(graphics)
```

```
for(i=1; i<=n; i=i+1){
```

```
    call segmtz(x(i)+dx(i), y(i), x(i)-dx(i), y(i), 1)
```

```
    call segmtz(x(i), y(i)+dy(i), x(i), y(i)-dy(i), 1)
```

```
}
```

```
QUERY( srt(oldsrt), adj(oldadj), cxy(cx,cy) )
```

```
SPECIFY( srt(45), adj(0) )
```

```
for(i=1; i<=n; i=i+1)
```

```
    call textz(x(i)+cx,y(i)+cy,TEXT(id(i)))
```

```
SPECIFY( srt(oldsrt), adj(oldadj) )
```

```
return
```

```
end
```

The interface routine can then call *cross2* to carry out the actual plotting.

The *CHAIN* statement takes more of the burden from the graphical algorithm by allowing another function to label the coordinate axes. The statement

```
CHAIN(axes,PAR,FILTER)
```

causes the *axes* function to draw the coordinate axes and surrounding box, and also to inherit the user-specified parameter values, and any arguments that this function did not use.

The final version of the *cross* interface routine is

```

FUNCTION cross(&)
STRUCTURE(x,y)
PLOTARGS
ARG(
    dx    /REAL/
    dy    /REAL/
    id    /CHAR/
    PAR
    &     # allows main, xlab, ...
)

if(LENGTH(x) != LENGTH(y) | LENGTH(x) != LENGTH(dx)
   LENGTH(x) != LENGTH(dy) | LENGTH(x) != LENGTH(id))
    FATAL(Lengths do not match)

call crange(x,dx,LENGTH(x),xmin,xmax)
call crange(y,dy,LENGTH(y),ymin,ymax)
SETUP(xmin,xmax,ymin,ymax)

call cross2(x,y,LENGTH(x),dx,dy,id)

CHAIN(axes,PAR,FILTER)
END

```

Notice that a check on the lengths of the arguments has also been included.

The most useful graphical algorithm for the S environment, then, is one that assumes that a coordinate system and axes have been set up by the interface routine. The algorithm draws the desired plot, and the interface routine *CHAINS* to function *axes* to label the plot.

In a stand-alone environment (outside of S), a graphical algorithm still needs to set up and draw axes. Therefore, when an algorithm is to be used both in and out of S, it is often constructed in two layers. The outer layer, used in stand-alone mode, sets up the axes, calls the inner routine, and labels the axes. The inner layer actually constructs the plot. When used in S, only the inner algorithm is called, with the facilities of the interface language replacing the outer routine. The stand-alone version of *cross* becomes


```

#cross scatter plot with crosses at  $x \pm dx$ ,  $y \pm dy$ 
subroutine cross(x,y,n,dx,dy,id)
integer n; real x(n),y(n),dx(n),dy(n)
POINTER id(n)

INCLUDE(graphics)

call beginz

call crange(x,dx,n,xmin,xmax)
AXIS(BOTTOM,xmin,xmax)

call crange(y,dy,n,ymin,ymax)
AXIS(LEFT,ymin,ymax)

call cross2(x,y,n,dx,dy,id)

call saxisz(BOTTOM,TRUE,TRUE); call saxisz(TOP,TRUE,FALSE)
call saxisz(LEFT,TRUE,TRUE); call saxisz(RIGHT,TRUE,FALSE)

call mtextz(BOTTOM,4.,TSTRING(Cross Represents Error for Each Observation))
call boxz

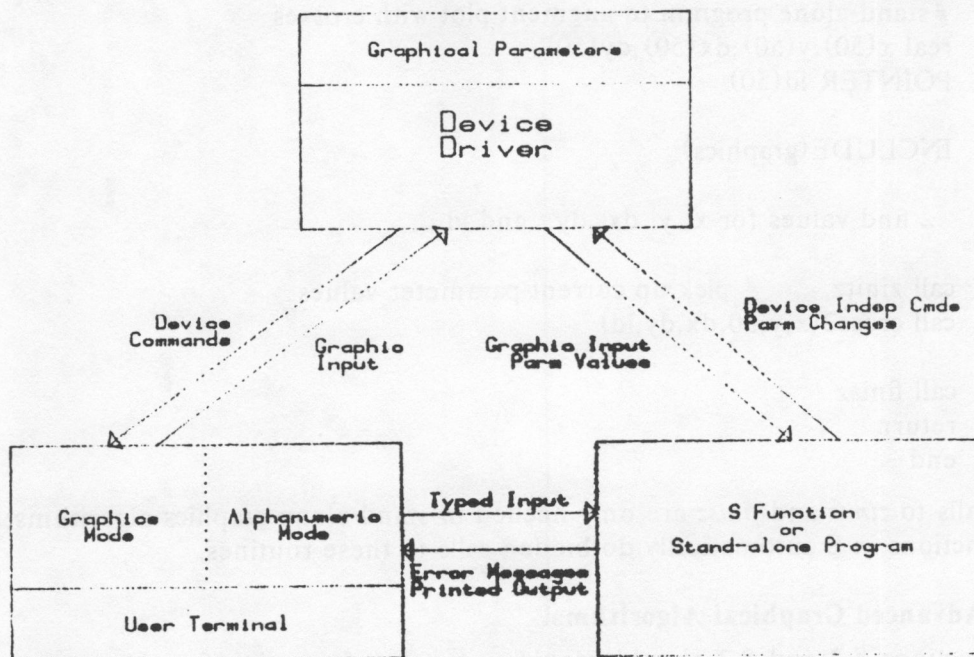
call finisz
return
end

```

8.3.2. Algorithms that Augment a Plot.

In both the S and stand-alone environments, all graphical operations are carried out through a *device driver* that knows how to control a specific device. This operation is illustrated by the following figure. Once it is activated, a device driver communicates with all graphical functions and also maintains the current state of the graphical parameters. Because a device driver may interact sequentially with a number of different graphical algorithms, it is possible for one algorithm to set the graphical parameters and for another algorithm to make use of these new parameter settings in later plotting. This is especially useful when one algorithm sets up a user coordinate system, and another algorithm augments that plot using the same coordinate system.

An S function that uses the algorithm *cross2* to augment an existing plot with crosses is



```

#add crosses to existing plot
FUNCTION addcross(&)
STRUCTURE(x,y)
PLOTARGS
ARG(
    dx    /REAL/
    dy    /REAL/
    id    /CHAR/
    PAR
)
if(LENGTH(x) != LENGTH(y) ... # as before

call cross2(x,y,LENGTH(x),dx,dy,id)

END
  
```

Notice how neatly the inner layer of our high-level algorithm can be used for this purpose.

To pick up the current values of the graphical parameters from the device driver but without forcing a new figure, a stand-alone algorithm can replace the call to *beginz* by

```
call zinitz
```

Therefore, a stand-alone main program to augment an existing plot with crosses would look like


```

#stand-alone program to augment plot with crosses
real x(50),y(50),dx(50),dy(50)
POINTER id(50)

INCLUDE(graphics)

... find values for x, y, dx, dy, and id ...

call zinitz      # pick up current parameter values
call cross2(x,y,50,dx,dy,id)

call finisz
return
end

```

The calls to *zinitz* and *finisz* are only needed in stand-alone graphics algorithms; graphical functions in S automatically do hidden calls to these routines.

8.4. Advanced Graphical Algorithms.

Sections 8.2 and 8.3 should provide enough information for the construction of simple graphical algorithms. It is usually best to develop simple algorithms initially, and to refine them later.

This section describes in much more detail the activities carried out by the graphical subroutines. It is recommended for advanced users.

8.4.1. Control of Figures, Plots, Margins.

A number of graphical parameters are devoted to positioning of figures on the output device and plots within the figure. This section describes these parameters, and related graphical subroutines.

As the second figure of section 8.1.1 illustrated, the device surface is divided into outer margins and figures, and each figure is divided into margins and a plot. Understanding this structure is important in writing algorithms that alter the size of any of these regions.

In general, graphical algorithms *should not* change parameters dealing with these regions. Instead, an ideal high-level algorithm should produce a suitable plot in the currently defined plot region, and label it in the current margins. In this way, the routine that calls the algorithm has control over the size and shape of the plot. The caller can also arrange to position the figures as desired on the page, without making changes to the called routines.

When writing graphical algorithms for use in S functions, the same considerations apply. Typically, the S user retains control over the parameters that define the figure and plot regions. These parameters can be changed in S by means of the *par* function.

Because it is desirable to allow a higher level routine to determine margin size and figure placement, the parameters and subroutines described in this section should ordinarily be used only by algorithms that produce a number of figures or those with special needs. With this warning in mind, we now describe the parameters.

In the simple case, there is one figure on the device surface. By default, the figure occupies the entire surface. Within the figure are the margins. They are controlled by the parameters

```
mar(bottom,left,top,right)
mai(bottom,left,top,right)
```

The second, *mai*, refers to margin size in inches. The first, *mar*, measures the size of the margins by the number of lines of text that can be accommodated. The parameter *mex(size)* is a measuring unit for the margins which describes the size of these lines of text, relative to the standard character height (actually, the inter-line spacing) on the device. Parameter *mex* does not affect character size, it only changes the way in which the margin is allocated and addressed. Thus

```
SPECIFY( mex(1.5), mar(3,4,5,1) )
```

allocates enough bottom margin to accommodate 3 lines of text of size 1.5, etc. The statement

```
SPECIFY( mar(4.5, 6, 7.5, 1.5) )
```

allocates the same amount of margin space but does not change the units of the margin coordinate system which are measured in lines of *mex*-sized text. The margin coordinate system will be described further in section 8.4.2.

Once the space for margins is reserved in the figure region, the remainder of the space may be used for the plot. The shape of the plot depends on the parameter

```
pty(type)
```

where *type* can be "s" for square or "m" for maximum size within the confines of the margins (the default). For special applications, the plot can also be specified in inches, by its aspect ratio, or as a fraction of the figure region; see the detailed documentation for routines *inpltz*, *arpltz*, and *drpltz*.

The situation becomes more complicated when more than one figure appears on the device surface. In this case, outer margins are normally allocated at the extremes of the device surface. The size of the outer margins is controlled by parameters

```
oma(bottom,left,top,right)
omi(bottom,left,top,right)
```

which, like their margin counterparts *mar* and *mai* specify outer margins in lines of *mex*-sized text or inches. By default, the outer margins are all zero. Outer margins can also be used when there is only one figure per page: for example, to provide consistent labelling for several pages, some with one figure per page and others with more than one.

Figures are allocated from the region not used for the outer margins. Most commonly, users want a regular array of figures on the device. The routines

```
call mfrowz(rows, cols)
call mfcolz(rows, cols)
```

set up a *rows* by *cols* array of figures, with *mfrowz* specifying that they are to be filled row-by-row, and *mfcolz* column-by-column. When in this mode, the parameter

```
mfg(i,j,m,n)
```

describes that the current figure is the *i*th row and *j*th column of an *m* by *n* array. This parameter can also be specified to begin plotting at a specific figure of the array. After the use of *mfrowz* or *mfcolz*, calls to *beginz* advance to the next clear figure in the array of figures.

Precise control over figure placement is available by subroutine *drfigz*, which directly specifies figure placement as a fraction of the device surface. In all of the above cases, parameter *fty* holds the type of the current figure, and the value "r" is multiple by rows, "c" is multiple by columns, "m" is maximum size, "d" is directly specified.

At any given moment, the graphical subroutines attempt to keep the graphical parameters consistent with one another. For example, if a change is made to the outer margins, the following sequence of operations takes place:

- 1 The size of the outer margins is subtracted from the size of the device.
- 2 The current figure is positioned within the remaining area according to the figure type and the *mfg* parameters.
- 3 The size of the margins is computed from parameters *mar* and *mex* and the known size of a standard character, and the margins are thus allocated within the figure.
- 4 The plot is recomputed based upon plot type *pty*.

Changes to figure parameters start the computation at step 2. Changes to margin parameters cause computations to begin at step 3.

At any intermediate stage, the set of parameter values may be inconsistent. For example, suppose that we want to set up a 3 by 3 array of figures and try

```
call mfrowz(3,3)
```

In step 2 above, the current figure will have an area of 1/9 of the device surface. Then, margins will be allocated from this area. If the margin size has not been changed from its default value, there may be no room in this small figure for all of the margins, thus causing an error. This problem can be solved by cutting down on margin size before changing to multiple-figure mode, and increasing margin size only after returning to one figure per page. The rule of thumb is to make changes which would increase the size of the plot (or decrease the margin size) first.

Because the plotting region may be changed when related parameters are changed, figure or plot defining parameters should not be reset if they are changed by an algorithm. Resetting the parameters would recompute the plot region, etc., and thus prevent further algorithms from augmenting the plot.

8.4.2. Margins and Outer Margins.

As described in section 8.1.2, text in the margin region is addressed by means of a margin coordinate system. This system uses three values to position text: *side*, *line*, and one *user coordinate*. The *side* value is either BOTTOM, LEFT, TOP, or RIGHT. For text positioned parallel to the side of the plot, the *line* value measures the distance from the edge of the plot to the near edge of the text. This measurement is done in units of *mex*-sized characters; i.e., if parameter *mex* is 2, then the units of the coordinate system are based upon the height of a twice-normal character and *line* 2.5 is two-and-a-half of those units from the edge of the plot. The user coordinate measures distance along the specified side.

This system provides a flexible means of addressing the margin region. The routine

```
call mtextz(side,line,text)
```

is most often used to plot text in the margin. It utilizes the parameter *adj* to control

the user coordinate position along the side: if *adj* is .5, the text is centered at the middle of the side, 0. left justifies at the minimum user coordinate, and 1. right justifies at the maximum user coordinate. The text is always plotted parallel to *side*.

A somewhat more general routine is

```
call gtextz(side,line,coord,text)
```

which plots *text* at the specified margin coordinate using the current values of string rotation and adjustment.

By default, the operation of *clipping* is carried out at the edge of the plotting region. Any points, text, or lines that extend into or beyond the margin are truncated at the point where they leave the plotting region. This prevents graphical operations from cluttering the margin, and can be used to keep outlying points off a plot. Clipping can be modified to allow plotting to eXPanD into the margin by means of parameter *xpd*. If *xpd* is TRUE, clipping will be done at the edge of the figure, rather than the edge of the plot. This is particularly useful for graphical functions that augment current plots, like the *addcross* function presented in 8.3.2. To allow added crosses to extend into the margin,

```
logical oldxpd
...
QUERY( xpd(oldxpd) )
SPECIFY( xpd(TRUE) )
call cross2( ... )
SPECIFY( xpd(oldxpd) )
```

The outer margins can be addressed identically to the margins. Routine *otextz* plots in the outer margin in a way analogous to *mtextz*. More generally, the parameter

```
omo(on)
```

controls operations in the outer margin. If *on* is TRUE, any operations that ordinarily would have occurred in the margin are instead done in the outer margins. In addition, parameter *omo* preserves the user coordinate system, so that data specified in user coordinates appears in the correct place. This means that axes can be plotted in the outer margins by *saxisz*. For example, when plotting a number of figures on the same page, all having the same x axis, a single set of axis labels could be placed at the bottom of the page:

```
SPECIFY( mar(0,5,1,5), oma(5,0,5,0) )
call mfrowz(4,1)
... produce 4 figures separated by only 1 line of TOP margin ...
... do not use saxisz to draw any x axes ...
SPECIFY( omo(TRUE) )
call saxisz(BOTTOM,TRUE,TRUE) #label x axis on bottom plot
```

8.4.3. Parameters of Physical Size.

Since the device driver knows the physical characteristics of the graphical device, it is possible to QUERY parameters


```
pin(width,height)
fin(width,height)
din(width,height)
```

which give the width and height of the plot and figure regions, and the device surface, respectively.

Another useful parameter is

```
uin(ux,uy)
```

which gives the number of inches corresponding to one user coordinate unit in both the x- and y-directions. This parameter is often used when physical relationships are to be preserved or when plotting is to be done in a specific distance. For example, to position a character string 1 inch to the right and 1 inch above the point (x,y)

```
QUERY( uin(ux,uy) )
text( x + 1./ux, y + 1./uy, TSTRING(My Label) )
```

It is also important occasionally to ensure that the x- and y-values of *uin* are equal. In drawing a picture of an object, it is appropriate to scale the size of the object to fit on the page, but not to change the x and y scales independent of one another. To set up a coordinate system addressed in inches,

```
QUERY( pin(px,py) )
SPECIFY( usr(0,px,0,py) )
```

To make sure that the x- and y-axes are commensurate, it is possible to make the plot type parameter square, and set up the coordinates identically on each axis:

```
call rangev(x,n,umin,umax)
call ranged(y,n,umin,umax)
SPECIFY( pty("s"), usr(umin,umax,umin,umax) )
```

8.4.4. Setting-up Coordinate Systems and Axes.

Once a plot and figure have been set up, it is normally necessary to establish a user coordinate system. The default user coordinate system places the point (0,0) at the lower left hand corner of the plot region, and (1,1) at the upper right. These default coordinates are sometimes reasonable to use in graphical algorithms, but most often, the algorithm will be more natural if an application-specific coordinate system is used.

If the algorithm knows the exact range of the user coordinates and there will be no need to generate axes, the user coordinates can be specified directly:

```
SPECIFY( usr(xmin,xmax,ymin,ymax) )
```

Be sure not to try to draw axes with *saxisz*, etc., since the axis parameters will not be correctly defined.

Since most applications require the plotting of coordinate axes, section 8.2.2 introduced the *AXIS* statement to set up a linear axis and coordinate system easily. The remainder of this section will present more powerful routines for establishing axes, and a detailed account of their operation.

First, however, we should mention several graphical parameters that affect axes. For example,

tck(size)

determines the length of axis TiCK marks as a fraction of the plot size. The default value is 0.02; interesting variations include negative values (ticks point outside of plot) and the value 1.0 (tick marks become grid lines).

Parameter

lab(nx,ny)

suggests the number of intervals that should appear on axes set up by high-level routines. Parameter *las* tells how axis labels should be oriented (0 = parallel to the axis, 1 = always horizontal) and *mgp* positions the axis line and labels in the margin.

The remainder of this section is very advanced, and can be ignored by most readers. Subroutines which set up the coordinate system and coordinate axes at the same time are:

call stdaxz(side,style,umin,umax,nint,flag)

call logaxz(side,style,umin,umax,nint,flag)

call timaxz(side,style,umin,umax,nint,flag)

to set up standard (linear) axes, logarithmic axes, and time axes, respectively. Note that the *x* and *y* axes are set up in separate calls, allowing arbitrary combinations of standard, logarithmic and time axes.

Argument *side* is BOTTOM, TOP, LEFT, or RIGHT. (BOTTOM and TOP refer to *x* axes, LEFT and RIGHT to *y* axes). The arguments *umin* and *umax* are the minimum and maximum user coordinates desired on the axis, and there should be approximately *nint* intervals labelled along the axis.

Before describing the *style* and *flag* arguments to these routines, it will be valuable to discuss how labels are chosen for axes. In general, a plotted axis is labelled with "pretty" numbers; namely, numbers which are 1, 2, or 5 times a power of ten. The distance between adjacent labels is another "pretty" number.

The axis routines operate by first finding a "pretty" number which is near the value $(umax-umin)/nint$. This value becomes the distance between axis labels. Then, pretty numbers for the ends of the axis are chosen. The *style* parameter controls this choice. In most cases, the axis is labelled at values more extreme than *umin* and *umax*. For example, if these arguments were 1.357 and 7.68, the axis might range from 0 to 8 in steps of 2, or from 1 to 8 in steps of 1. This "standard" axis labelling is used if *style* is "s". It may also be desirable to make maximal use of the page for the plotted data. In this case, an "internal" axis can be used. Here, pretty values of the axis extremes are chosen *within* the values of *umin* and *umax*. In the example, the axis could be labelled from 2 to 7 in steps of 1.

The internal axis style brings up an important point: the axis limits and the corresponding coordinate system limits are related, but are not necessarily identical. For an internal axis, the coordinate system ranges from *umin* to *umax*, but the axis is labelled at pretty values within that range.

Another axis style is "extended". The axis labels are identical to those of a standard axis, but the user coordinates are extended to ensure that neither *umin* nor *umax* lie right at the user coordinate limit. This prevents data from being plotted directly on the box surrounding the plot. If *style* is given as "", then the previous style is used. This again gives the higher-level routine a measure of control over the lower-level operations. The default style is "e".

Logarithmic and time axes are parameterized in a way similar to linear axes. In all, each axis is described by

- 1 *minimum* and *maximum* user coordinates (parameter *usr*).
- 2 the *type* of axis: standard (linear), logarithmic, or time (parameters *xaxt* and *yaxt*).
- 3 the *style* of axis labels: standard, internal, or extended (parameters *xaxs* and *yaxs*).
- 4 a set of three parameter values telling where axis labels should be placed. For linear axes, these three values are the minimum axis label, the maximum axis label, and the number of intervals the axis contains. Logarithmic and time axes are also described by three values. (parameters *xaxp* and *yaxp*).

We can finally describe the *flag* argument to these routines. If *flag* is 3, all of the axis parameters are computed as outlined above, their values are saved in the graphical parameter array, the user coordinate system is modified, and the axis is then plotted. If *flag* is 2, the plotting is suppressed. (It is often suitable to produce a plot and label the axes last, thus giving the user time to look at the display while the axes are drawn.) Standard, logarithmic, and time axes can be plotted later by routines *saxisz*, *laxisz*, or *taxisz*. Finally, if *flag* is 1, everything but setting the user coordinates is performed. This is used for special applications, for example, where only part of the user coordinates are to have labelled axes. Another routine that performs this task is *diraxz*, described in the detailed documentation.

8.4.5. Summary of Graphical Parameters.

The following table gives a list of the mnemonic abbreviations for the graphical parameters, along with their default values. The parameters are grouped according to the types of graphical output they affect.

What	Units	par(Values)	Default
Characters.			
Size	Relative to standard	cex(size)	1
Size	Inches	csi(inches)	—
Size	Fraction of Plot	csr(fraction)	—
Size	User coords	cxy(width,height)	—†
String Rotation	Degrees ccw from horiz	srt(angle)	0.
Char Rotation	Degrees ccw from horiz	crt(angle)	0.
String Justification	0 =left, 1 =right, .5 =center	adj(pos)	.5
Lines.			
Line Type	device dependent	lty(type)	0 (solid)
Line Width	device dependent	lwd(width)	1
Other Symbols.			
Plotting Char	character	pch(char)	"**"
Mark Height	inches	mkh(inches)	same as std char
Circle Smoothness	rasters	smo(error)	1
Lines and Characters.			
Color	device dependent	col(color)	0
Margins.			

Outer Margins	lines of text	oma(bot,lef,top,rig)	0,0,0,0*
Outer Margin	inches	omi(bot,lef,top,rig)	0,0,0,0*
Outer Margins	fraction of device	omd(xmin,xmax,ymin,ymax)	0,1,0,1
Outer Margins On	logical	omo(on)	FALSE
Margins	lines of text	mar(bot,lef,top,rig)	7,7,7,7*
Margins	inches	mai(bot,left,top,rig)	—
Margin Units	fraction of standard	mex(size)	1
Plots and Figures.			
Figure Type	character	fty(char)	"m" (maximum size)
Multiple Figures	array m by n	mfg(i,j,m,n)	1,1,1,1
Clipping	logical	xpd(off)	FALSE
Plot Type	character	pty(char)	"m" (maximum size)
Box Type	character	bty(char)	"o"
New Plot	logical	new(empty)	TRUE
Axes.			
Axis Type	character	xaxt(char)	"s" (standard)
		yaxt(char)	
Axis Style	character	xaxs(char)	"e" (extended)
		yaxs(char)	
Axis Parameters	label info	xaxp(p1,p2,p3)	0,1,5
		yaxp(p1,p2,p3)	
User Coordinates		usr(xmin,xmax,ymin,ymax)	0,1,0,1
Label Style	0 =parallel, 1 =horiz	las(style)	0
Tick Length	fraction of plot	tck(leng)	0.02
Label Intervals	number	lab(nx,ny)	5,5
Axis Position	lines	mgp(lab,num,line)	3,1,0
Size Related.			
Device Size	inches	din(width,height)	—†
Figure Size	inches	fin(width,height)	—†
Plot Size	inches	pin(width,height)	—†
Coordinate Size	inches per unit	uin(ux,uy)	—†
Raster Size	inches	rsz(rx,ry)	—†
Miscellaneous.			
Error Mode	—1 =ignore, 0 =print, 1 =fatal	err(mode)	0
Device Code	device dependent	dev(code)	—†

Unless marked with †, all parameters can be both specified and queried. In general, the parameters listed in the table can be specified or queried with as many arguments as are of interest. Thus, for example, to specify only the x-coordinates

SPECIFY(usr(xmin,xmax))

or to QUERY only the y-coordinates

QUERY(usr(,ymin,ymax))

A few parameters (indicated by an "*" in the table) can not be abbreviated in this way for *SPECIFY*. The effect of parameters listed as "device dependent" can be found by looking at the detailed documentation for the S device driver functions.

8.4.6. Graphical Input.

A number of graphical devices provide some special form of *graphical input*. The way in which it is carried out differs from device to device: pen plotters normally have buttons or a joystick to allow the user to manually position the pen, scopes may have cursors, etc. For devices with no special mechanism, the device driver will ask the user to type in the desired coordinates. Details can be found in the detailed documentation for the S device driver functions. Whatever the mechanism, the device driver takes care of appropriate user interaction in response to the subroutine

```
call inputz(x,y,n,nmax)
```

and returns up to *nmax* different pen positions in vectors *x* and *y*. The value of *n* gives the number of points that were actually given by the user. The returned points are all in user coordinates.

This graphic input facility is used in the S functions *rdpen* and *identify*, and could be used in other algorithms to interact with a menu of commands, identify outliers, etc.

8.4.7. Debugging.

Debugging is often an easier process for a graphical algorithm than for a purely computational one. The ability to watch the interactive graphics being produced often provides clues to the origin of problems. Of course, any of the debugging techniques of section 6.1.7 and 7.1.5 can be used with graphical algorithms.

The values of the graphical parameters are often useful for debugging. They can be obtained by the S function *pardump* which takes as arguments the character string names of graphical parameters. Thus

```
pardump("usr","srt")
```

will give the current user coordinate and string rotation values. The function *pardump* with no arguments gives the values of all parameters.

A similar facility is available in subroutine form. Executing

```
call dumpaz
```

produces a list of the current values of all parameters.

8.5. Available Graphical Subroutines.

A number of high-level graphical subroutines are available for use in graphical algorithms. We have left them out of the discussion until now because most of these high-level routines are already available in the form of S functions. For example, high-level routines exist for contour-, scatter-, and box-plots.

barz	bar plot
bplotz	barocentric plot of mixture $x+y+z$ data
bxpz	box plots
contrz	contour plot
eepltz	empirical q-q plot
eqpltz	empirical-theoretical q-q plot
hhpltz	histogram
nplotz	scatter plot with repeated point counts
persp	perspective plot
piez	pie chart
plotl	very high-level controlled scatter plots

pltusa	map of USA
rplotz	robust scatter plot
splotz	scatter plot
tspltz	time-series plot

More interesting, perhaps, are the routines that underly these high-level subroutines. These often produce the essence of the plot, without dealing with details of axis construction, labelling, etc. For example, the routines *bxz* and *bxnz* draw a single standard or notched box for use in a boxplot. Underlying routines like this may be useful in the construction of graphical algorithms.

Some of the potentially useful routines are:

bhtchz	shades bars of bar chart
bmglgz	legend for shaded bar chart
bnamez	names for bars of bar chart
bxz	box to represent distribution
bxnz	notched box for distn with conf limits
cont2z	contour lines
hatch	shade in polygonal area
hdlinz	high-density vertical lines
hhbrkz	break points for histogram
hhdonz	number of classes for histogram
linesp	lines with transformation set up by pictur
lintfz	lines with transformed coordinate system
npntsz	points with counts of repeats
pictur	set up translation, scale change, rotation
pnttfz	points with transformed coordinate system
ptitle	titles
rpntsz	robust plot of scatter of points
shadez	shade in rectangle
titlez	titles
tsvecz	vector of x values for time-series plot
usabdy	coordinates of boundary of USA

8.6. Stand-Alone Graphical Algorithms.

The graphical subroutines, when used in a stand-alone mode, provide the capability for creating graphical tools, i.e., programs that can be used together to construct plots. There are a few of these tools already available; they can be invoked by

S GRAPH name

Most fundamental of these stand-alone tools are device drivers for the same devices supported by S. (In fact, they have the same names as the S device functions). Thus,

S GRAPH printer

initiates the device driver for printing devices. A stand-alone device driver sets the environment so that any successive graphics utilities communicate with the device driver, just as the current S device remains in effect once specified. Signalling end-of-file to the command interpreter causes the device driver to terminate.

Several stand-alone utilities are also available via *S GRAPH*. Most important is *pf*, a high-level plotting tool similar to, but more general than, the UNIX *graph* command. It reads (x,y) pairs from its standard input and produces a scatter plot. Control over the details of the scatter plot is provided by an argument that describes the operations to be performed. In general, *pf* is invoked as

S GRAPH pf control title subtitle xlabel ylabel

The program reads coordinate pairs from standard input, uses argument *control* to describe the desired plot, and takes titles and axis labels from the other arguments. Any trailing arguments may be omitted.

A typical *pf* command might be

S GRAPH pf logx/dashline "Main Title" "" "X Label" <iris

to produce a plot of the data in file *iris* connected by dashed lines on a logarithmic x axis.

Another utility is *rdpen*, which utilizes the graphic input facility of the device and writes the specified coordinate pairs on the standard output. Thus, *rdpen* and *pf* can be used together

S GRAPH rdpen | S GRAPH pf lines/over

to add a set of connected lines to the current plot. Further details of these utilities (and a complete list of *control* values) can be found in the detailed documentation.

Users can construct similar stand-alone facilities by use of the *S PROGRAM* utility, as described in section 8.3.

8.7. Device Drivers.

A number of device drivers are available in both the S and stand-alone environments. They support the Tektronix 4006 and 4010 scopes (tek10), Tektronix 4012 scope (tek12), Tektronix 4014 scope (tek14 and tek14q), Hewlett-Packard 7221 pen plotter (hp72h, hp72v, hp72f), Tektronix 4662 pen plotters (tek46h, tek46v, tek46f), and printer-like devices (printer).

This section describes the construction of device drivers for new graphical devices.

8.7.1. Organization of Device Driver Routines.

A device driver is constructed from a number of individual subroutines, each of which carries out a well-defined task. Specifically, there are routines to initialize, seek, draw a line, plot a character, eject to a new frame, read graphic input, reset the device to allow non-graphic output, and to wrap-up. All of these routines must be furnished, although not all of them need to actually do anything. For example, on a device with no distinction between graphic mode and terminal mode, the reset routine may be null.

In order to make them easy to implement, all of these routines operate in the *raster* or *device coordinate* system. These are the coordinates natural to the graphic device; most devices are addressed by integral values from 0 to some specified maximum along each axis.

The device driver routines are designed to support relatively dumb graphical devices. They do not assume that the device can perform coordinate transformations (mapping user coordinates to device coordinates), or clipping. Also, the device drivers do not assume capabilities for the device to plot a set of line segments, a number of points, or even a text string. Instead, all of these operations are carried out one line or character at a time.

This normally means that some unnecessary operations are carried out on more

intellegent devices, that can, for example, plot text strings. However, the overhead introduced by this low-level treatment of the device is typically not important in the overall context of interactive data analysis. Readers who must use the features of a new device can read the programs that invoke the device driver operations and make their own changes.

The initialization routine

```
zparmz(par,n)
```

is called once, before any other device driver routines are called. It has the responsibility to initialize the device and to set up certain values in the graphical parameter array. In particular, it should contain

```
INCLUDE(graphics)
```

and should define the following parameter values:

```
call rfill(0.,am,39)    # initialize first 39 locations of parameter
                        # array with zeroes
```

```
am(20) = cw # character width in rasters (including inter-char space)
am(21) = ch # character height in rasters (including inter-line space)
```

```
am(22) = xmin
am(23) = xmax          # min and max raster addresses for x-axis
```

```
am(24) = ymin
am(25) = ymax          # min and max raster addresses for y-axis
```

```
am(26) = dx
am(27) = dy # character addressing offset (see below)
```

```
am(28) = rw # raster width and height in inches
am(29) = rh # typically am(28) = (page width)/(am(23)-am(22)), etc
```

```
am(30) = dev # device code number
```

```
am(1) = flag # if 0, device cannot change character size
am(31) = flag # if 0, device cannot rotate characters
```

These parameter values can usually be obtained easily from the programmer's manual for the graphic device. The only non-obvious parameters are *am(26)*, *am(27)*, that tell where a character is addressed on the current device. For example, the values (0,0) indicates that the lower-left of the character is the addressed position, (.5,.5) that characters are centered at the current device position, etc. Sometimes the correct values of these parameters may be obtained by experimentation; try plotting a large character atop a cross and note whether it is centered.

Certain of these parameters are somewhat rough indicators of the capabilities of the device. If the character size change or rotation parameters indicate that the device has these capabilities, it is assumed that both parameters are infinitely variable, although many devices have only a discrete number of sizes and rotations.

The argument *par* provides a list of *n* device-specific parameters. In many cases, *n* may be zero. For certain devices, these parameters can describe device

characteristics such as width, height, etc. See section 8.7.4.

The basic plotting operations on the device are carried out by analogy with a pen-plotter device. The routine

```
zseekz(x,y)
```

causes a *seek* operation, i.e., the device positions itself (without drawing anything) to the coordinates (x,y). (Remember, these are integer raster coordinates).

The device driver can now either draw a line from its current position to a new position, or it can plot a character at the current position. These operations are carried out by routines

```
zlinez(x,y)
zptchz(char,rot)
```

The character plotting routine should interpret the argument *rot* as a rotation angle for the character, measured in degrees counter-clockwise from horizontal.

Most often, the *zseekz* routine takes responsibility for controlling changes of color or line type if the device has those capabilities. Routine *zptchz*, in addition to changing character rotation, also should set character size according to the *cex* parameter.

Routine *zejecz* is called whenever it is necessary to advance to a new frame on the graphic device. On many devices, it will issue a message to the user such as "GO?", which allows the user time to put a new sheet of paper in a plotter or to make a copy of information on a scope. Once a reply is received to the message (routine *zoutrz* section 8.7.2), it clears the screen, etc.

Some devices provide a distinction between *alphanumeric* mode and *graph* mode. Others, e.g. pen plotters, may be turned on or off to enable or disable graphics. In either case, it is important that, at the conclusion of graphic output, the device be left in a mode suitable for normal terminal interaction. This is the responsibility of routine

```
zflshz
```

which flushes any remaining graphic commands out to the device and restores alphanumeric mode. Routine *zseekz* is always the first routine called to resume plotting, and it should take responsibility for enabling graph mode.

Some care should be taken to make the alphanumeric/graph mode switching reasonably efficient. In a system of graphical subroutines, it is not possible to know when the calling routine will do alphanumeric output to the terminal. So, to be on the safe side, the graphical subroutines return the terminal to alphanumeric mode whenever they give control back to the caller. This may mean lots of mode switching.

When the device driver is called for the final time, *zwrapz* is invoked to wrap-up operations. In many cases, nothing special is necessary for wrap-up, although pen plotters may put away the pen, etc.

Reading the source code for current device drivers is perhaps the best way to get a feel for how a new device driver should be implemented.

8.7.2. Portability Considerations.

Three routines are provided in the support library to enhance the portability of device driver code. Most interactive devices are prepared to receive ASCII characters from a computer system, and to recognize special combinations of these characters as graphic commands. Unfortunately, algorithms written in a FORTRAN-based language have no capability to communicate some of these characters to the device. The routines

```
call zoutrz(buf,n)
call zoutwz(buf,n)
ich = i6to9z(char)
```

are designed to remedy this situation. The first two take an integer vector *buf* and interpret its contents as coded ASCII characters. For example, 12 is the code for the "form-feed" character. The first *n* values in *buf* are sent to the terminal as *n* ASCII characters, with no appended carriage return, etc. In addition, *zoutwz* waits until some input is received from the terminal before returning (normally used when the user is prompted for permission to clear the device and go to the next frame.)

The function *i6to9z* translates a single character from the machine's internal code to the ASCII code used by *zoutrz* and *zoutwz*. It is normally used by *zptchz* to turn characters into ASCII for the output device.

By using these routines to communicate with a graphic device, it is possible to be certain of the ASCII characters that will reach the device, and yet avoid any knowledge of the machine's internal character set.

8.7.3. Control of Graphic Input.

One of the most difficult parts of writing a device driver is providing for graphic input. All graphic input is done by the user-written routine

```
zquxyz(x,y,indic)
```

where (x,y) is the set of device coordinates for the selected point, and *indic* is an indicator that can be used to show status information. One special use of *indic* is that negative values signal end of graphic input; positive values can be used to code pen status (up or down), current color, character used to transmit input, etc.

One of the most difficult tasks with graphic input is controlling line turnaround, full/half duplex transmission (echoed characters may interfere with the device), etc. Because they are tailored to the environment, graphic input routines are typically non-portable.

8.7.4. Writing a Device Driver for S.

To create an S device driver function, use the "-d" flag on the *FUNCTION* command. For example, if we acquire a new Zork-7000 graphics device and want a device function *zork*,

```
S FUNCTION -d zork zork.i
```

With this flag, the utility does a number of things. A directory named *zorksource* is created to hold the source files for the device driver routines (*zparmz*, etc.). In addition, provision is made for two new S functions: *zork* and *dev.zork*.

To create the device driver, write the device dependent routines as outlined in

section 8.7 and place them on files in directory *zorksource*.

Next, write an S Function *zork.i* as follows:

```
# Driver for zork7000 graphics terminal
FUNCTION zork()
DEVICE_DRIVER
END
```

Once the interface routine is written, use the *MAKE* utility to create executable versions of the interface routine and the associated device driver routine.

```
S MAKE zork dev.zork
```

All routines in the device source directory are compiled and loaded in the *dev.zork* routine.

The interface routine can serve a more interesting purpose if *zparmz* allows optional parameters. In this case, if the interface routine returns a vector named *parms*, it will be passed on to *zparmz*. For example, if our Zork-7000 is a pen plotter with variable paper size,

```
# Driver for zork7000 pen-plotter
FUNCTION zork(
    width /REAL,1,10./
    height /REAL,1,8./
)
STRUCTURE(parms/REAL,2/)
parms[1] = width
parms[2] = height
DEVICE_DRIVER
RETURN(parms)
END
```

Routine *zparmz* will be called with a vector containing the two parameters. It can use this information to set up an appropriate device coordinate system, character size, etc. This will allow users to invoke the *zork* device function with optional parameters specifying the paper width and height.

Function Detailed Documentation

Note: This documentation reflects recent changes to S; these changes may not have been made at certain installations. If a function behaves in a manner inconsistent with this documentation, consult the on-line documentation, available via the *help* function. This describes the versions of functions at any particular installation.

Function Documentation Table of Contents

%%	(see arith)	c	
%/	(see arith)	%c	(see matpt)
*	(see arith)	C	(see PROGRAM)
**	(see arith)	cancor	
+	(see arith)	cauchy	
-	(see arith)	cbind	
/	(see arith)	ceiling	
~	(see arith)	CHAPTER	
->	(see assign)	chapter	
<-	(see assign)	chisq	
-	(see assign)	chol	
!=	(see logic)	chull	
<	(see logic)	code	
<=	(see logic)	coerce	
=	(see logic)	col	
>	(see logic)	compname	
>=	(see logic)	contour	
!	(see logicop)	cor	
&	(see logicop)	cos	
	(see logicop)	ctr	
%*	(see matp)	cumsum	
%	(see operator)	cut	
\$	(see select)	cutree	
:	(see seq)	cycle	(see time)
[	(see subset)	defer	
]	(see subset)	define	
abline		density	
abs		detach	
acos		devices	
again	(see edit)	dget	
all		diag	
any	(see all)	diary	
aperm		diff	
append		discr	
apply		dist	
approx		dprompt	
arith		dput	
array		dump	
arrows	(see segments)	edit	
asin	(see acos)	EDITDOC	(see PROMPT)
assign		eigen	
atan		else	(see syntax)
attach		encode	
axes	(see title)	end	
backsolve		exp	
barplot		extract	
BATCH		f	
beta		faces	
box		floor	(see ceiling)
boxplot		for	(see syntax)
BPLOT		FORTRAN	(see PROGRAM)
bxp		frame	

FUNCTION	(see CHAPTER)	
gamma		mean
gammadist		median
get		medit
glim		min (see max)
gs		mode
hardcopy		mprint
hatch		mprompt
hclust		mstr
help		mstree
hist		mtext
hp72		mulbar
hp7220h	(see hppl)	na
hp7220v	(see hppl)	NA (see na)
hp7225	(see hppl)	napsack
hp72h		ncol
hp72v		ncomp
hppl		NEWDOC (see PROMPT)
identify		norm
if	(see syntax)	nper (see end)
ifelse		nproc
interp		nrow (see ncol)
knapsack	(see napsack)	%o (see outer)
llfit		operator
labclust		option (see options)
lag		options
leaps		order
len		outer
lgamma	(see gamma)	pairs
lines		par
list		pardump
lnorm		pbeta (see beta)
log	(see exp)	pcauchy (see cauchy)
log10	(see exp)	pchisq (see chisq)
logic		persp
logicop		pf (see f)
logistic		pgamma (see gammadist)
loglin		photo
logo		pie
lowess		plclust
ls	(see list)	plnorm (see lnorm)
%m	(see coerce)	plgis (see logis)
macros		plot
mail		plotfit
MAKE	(see CHAPTER)	pmax
match		pmin (see pmax)
matlines	(see matplot)	pnorm (see norm)
matplot		points (see lines)
matpoints	(see matplot)	ppoints
matrix		prcomp
matrix-cross	(see matpt)	precedence
matrix-multiply	(see matp)	prefix
max		pretty
mdscale		print
		PRINTDOC (see PROMPT)

printer		save	
prism	(see photo)	scale	
prod		search	
PROGRAM		segments	
PROMPT		select	
pt	(see tdist)	seq	
punif	(see unif)	show	
q		sin	(see cos)
qbeta	(see beta)	sink	
qcauchy	(see cauchy)	smatrix	
qchisq	(see chisq)	smooth	
qf	(see f)	solve	
qgamma	(see gammadist)	sort	
qlnorm	(see lnorm)	source	
qlogis	(see logis)	split	
qnorm	(see norm)	sqrt	(see exp)
qqgamma	(see qqplot)	stab	
qqnorm	(see qqplot)	stare	(see photo)
qqplot		stars	
qt	(see tdist)	starsymb	
qunif	(see unif)	start	(see end)
range		stem	
rank		structure	
RATFOR	(see PROGRAM)	subset	
rbeta	(see beta)	subtree	
rbind	(see cbind)	sum	(see prod)
rbiwt		svd	
rcauchy	(see cauchy)	sweep	
rchisq	(see chisq)	symbols	
rdpen		syntax	
read		sys	
reg		System	
regprt		t	
regress		table	
regsum		tbl	
rep		tdist	
replace	(see append)	tek10	(see tek12)
replot		tek12	
restore		tek14	(see tek12)
rev		tek14q	(see tek12)
rf	(see f)	tek46	
rgamma	(see gammadist)	tek46h	
rlnorm	(see lnorm)	tek46v	
rlogis	(see logis)	text	
rm		time	
rnorm	(see norm)	title	
round		tprint	
row	(see col)	trunc	(see ceiling)
rreg		ts	
rstab	(see stab)	tslines	(see tsplot)
rt	(see tdist)	tsmatrix	
runif	(see unif)	tsplot	
sample		tspoints	(see tsplot)
sapply		twoway	

unif
uniq (see unique)
unique
usa
var (see cor)
vu
window
write
xor

abline: Plot Line in Intercept-Slope form

```
abline(reg)
abline(coef)
abline(a, b)
abline(h=)
abline(v=)
```

ARGUMENTS:

reg: a regression structure; i.e., *reg* is a structure that contains a component *coef* giving coefficients of a fitted line (with intercept term first).

coef: vector containing the intercept (a) and slope (b) of the line $y=a+b*x$

a,b: intercept and slope as above.

h: vector of y-coordinates for horizontal lines across plot.

v: vector of x-coordinates for vertical lines across plot.

Graphical parameters may also be supplied as arguments to this function (see *par*).

The effect of *abline* is that the line $y=a+b*x$ is drawn across the current plot (where there is any intersection of the line and plot; a warning is given if the line does not intersect the plot).

EXAMPLES:

```
abline(0,1)    # draws line with 0 intercept and slope 1
abline(l1fit(x,y)) # line produced by L1 fit to x and y
abline(v=c(0,10)) # vertical lines at x==0 and 10
```

abs: Absolute Value

```
abs(x)
```

ARGUMENTS:

x: numeric structure. Missing values (NAs) are allowed.

VALUE:

numeric structure like *x*, with absolute value taken of each data value.

acos asin: Inverse Trigonometric Functions

```
acos(x)
asin(x)
```

ARGUMENTS:

x: numeric structure. Missing values (NAs) are allowed.

VALUE:

structure with angles in radians. For values of *x* outside the interval $[-1,1]$, NA is returned and a warning is given. The range of the result is $0 \leq \text{acos}(x) \leq \pi$ and $-\pi/2 \leq \text{asin}(x) \leq \pi/2$.

again: (see `edit`)

all any: Logical Sum and Product

`all(x)`
`any(x)`

ARGUMENTS:

x: Logical expression. Missing values (NAs) are allowed.

VALUE:

either TRUE or FALSE. *all* evaluates to TRUE if all the elements (other than NAs) of the argument are true; FALSE otherwise. *any* evaluates to TRUE if any of the elements of the argument are true; FALSE otherwise.

EXAMPLES:

`if(all(x>0))x ← sqrt(x)`

any: (see `all`)

aperm: Array Permutations

`aperm(a, perm)`

ARGUMENTS:

a: array to be permuted. Missing values (NAs) are allowed.

perm: vector containing a permutation of the integers 1:n where n is the number of dimensions in the array *a*. The old dimension given by *perm[j]* becomes the new *j*-th dimension.

VALUE:

array like *a*, but with the observations permuted according to *perm*, e.g., if *perm* is *c(2,1,3)*, the result will be an array in which the old second dimension is the new first dimension, etc.

If *a* contains a component *Label* (as in tables constructed with the *table* function), it will also be permuted.

append replace: Data Merging

`append(x, values, after)`
`replace(x, list, values)`

ARGUMENTS:

x: vector of data to be edited. Missing values (NAs) are allowed.

list: indices of the elements in *x* to be replaced.

values: vector of values to replace the list of elements, or to be appended after the element given. If *values* is shorter than *list*, it is reused cyclically. Missing values (NAs) are allowed.

after: index in *x* after which *values* are appended. *after*=0 puts *values* at the beginning.

VALUE:

the edited vector, to be assigned back to *x*, or used in any other way. Remember, unless the result is assigned back to *x*, *x* will not be changed.

EXAMPLES:

```
x ← replace(x,3,1.5)    # Replace x[3] with 1.5
x ← append(x,c(3.4,5.7),6)  # Append two values in x[7],x[8]
replace(x,c(3,7,8,9),0)    # Replace the four elements with 0
# Alternatives:
x[-(1:5)]                # Returns all but first five values
x[3]←1.5                 # Replaces x[3] with 1.5
                        # same as x←replace(x,3,1.5)
```

apply: Apply a Function to Sections of an Array

```
apply(a, margin, fun, arg1, arg2, ...)
```

ARGUMENTS:

- a:** array. Missing values (NAs) are allowed.
- margin:** the subscripts over which the function is to be applied. For example, if *a* is a matrix, 1 indicates rows, and 2 indicates columns. For a more complex example of the use of *margin*, see the last example below. In general, a subarray is extracted from *a* for each combination of the levels of the subscripts named in *margin*. The function *fun* is invoked for each of these subarrays, and the results, if any, concatenated into a new array. Note that *margin* is also the dimensions of *a* which are retained in the result.
- fun:** character string giving the name of the function to be applied to the specified array sections.
- argi:** optional, any arguments to *fun*; they are passed unchanged.

VALUE:

array whose dimensions are defined by *margin*. If the result of each application of *fun* has a length that is greater than 1, this length will be the new first dimension of the result. If *margin* only specifies one dimension to be saved, and *fun* returns results of length 1, a vector will be returned.

The system macros *?row(a,fun,...)* and *?col(a,fun,...)* apply a function over rows and columns of a matrix.

EXAMPLES:

```
apply(x,2,"mean",trim=.25) # 25% trimmed column means
                        # The result is a vector of length ncol(x)
apply(x,2,"sort")         # sort columns of x
apply(z,c(1,3),"sum")
```

In the last example, *z* is a 4-way array with dimension vector (2,3,4,5). The expression computes the 2 by 4 matrix obtained by summing over the second and fourth extents of *z* (i.e., *sum* is called 8 times, each time on 15 values).

Each section of the input array is passed as the first argument to an invocation of *fun*. It is passed without a keyword modifier, so, by keywords attached to *argi*, it should be possible to make the array section correspond to any argument to *fun*.

The function must return the same number of values (possibly 0) each time it is invoked on an array section.

System infix operators such as "+" can also be passed as functions.

sapply can be used to apply over all components of structures. The language looping construct *for* is more general than *apply*, but is less efficient computationally.

approx: Approximate Function from Discrete Values

```
approx(x, y, xout, method, n, rule)
```

ARGUMENTS:

- x,y:** pairs of points giving (x,f(x)). Can be a time series or a structure containing components named x and y.
- xout:** optional set of x values for which function values are desired. If *xout* is not specified, the function will be approximated at *n* equally spaced data points, spanning the range of x.
- method:** character describing the method to be used in approximating the function. Possible values are "linear" for linear interpolation (later may have "spline", etc).
- n:** optional integer giving the number of points evenly spaced between the minimum and maximum values in x to be used in forming *xout*. Default 50.
- rule:** integer describing the rule to be used for values of *xout* that are outside the range of x. If *rule* is 1 (the default), NAs will be supplied for any such points. If *rule* is 2, the y values corresponding to the extreme x values will be used.

VALUE:

structure with components named x and y.

EXAMPLES:

```
approx(x,y,newx) # linear interpolation at newx
```

+ - / %% %/ * ^ or ** : Arithmetic Operators

```
expr1 op expr2
```

ARGUMENTS:

expr: numeric structure. Missing values (NAs) are allowed.

VALUE:

numeric result, mode is real if either *expr* is real. The exception is integer divide (%/), which truncates its result, returning an integer.

The operator %% is the modulus function, "^" and "**" are synonyms for exponentiation.

See also *precedence*.

EXAMPLES:

```
x-mean(x)
(1+(5:8)/1200)^12 # compound interest, 5:8 per annum monthly
```


array: Create an Array

```
array(data, dim)
```

ARGUMENTS:

data: numeric vector containing the data values for the array in the normal array order: the first subscript varies most rapidly. Missing values (NAs) are allowed.

dim: integer vector giving the extent for each dimension. Default is *len(data)*.

VALUE:

an array with the same mode as *data*, and dimensionality described by *dim*. If *data* does not completely fill the array, it is repeated until the array is filled (giving a warning if it does not take an integral number of repetitions of *data*).

EXAMPLES:

```
iris←array(read("irisfile"),c(4,3,50))
      # creates a 4 by 3 by 50 array
```

arrows: (see *segments*)

asin: (see *acos*)

← → _ or assign: Assignment

```
name ← expression
name _ expression
expression → name
assign(name, expression, pos=)
```

The assignment operator, in any of its forms, causes the value of the expression to be saved on the working storage file under *name*. Note that the arrow always points toward *name*. The left-arrow is made up of the two characters less-than and minus. An underscore can also be used for left-arrow. Right-arrow is minus and greater-than.

Once an assignment is made, the data can be retrieved by mentioning the name used in the assignment. The value of the assignment (when used as an expression) is the *expression* part of the assignment.

ARGUMENTS:

pos=: The position in the current search list of the database on which *expression* is to be stored. Default 1. See *attach* and *save*. *save* is the usual way to assign, other than to the working data.

EXAMPLES:

```
y←sqrt(x←runif(100))    # 100 uniforms saved as x
      # square root of sample saved under the name y
assign("x",runif(100))  # equivalent to x←runif(100)
for(i in 1:6) assign(char[i],x[,i])
      # saves each column of matrix x under different name
```

atan: Inverse Tangent

atan(x)
atan(x, y)

ARGUMENTS:

- x: numeric structure giving tangents of desired angles.
y: numeric structure which along with x, determines the tangent of the desired angle as x/y . The two-argument version should be used if y may be near zero.

VALUE:

the angles in radians. Where both x and y are zero, the value is NA. The range of the value is $-\pi/2 \leq \text{atan}(x) \leq \pi/2$ and $-\pi < \text{atan}(x,y) \leq \pi$.

attach: Attach a New Database

attach(file, pos)

ARGUMENTS:

- file: character string giving the name of a new database to be accessed. The name must be a correct file-system directory reference, such as "/usr/abc/swork" for the working database in directory "/usr/abc".
pos: position in database search list that *file* should occupy. The databases originally in position *pos* or beyond are moved down the list after *file*. Default, after all current databases. If *pos*=1, *file* will be attached as the working database (you must have write permission).

When a user logs on, three databases are attached: working, save and shared. See the user guide, section 3.2 & 4.2. See *detach* for the procedure to detach a database from search list, and *search* to see the current list of attached databases. Databases are attached only for the current invocation of S. The only files that can be attached are those formatted as S database or working files; ordinary data files should be read by means of *read*; files produced by *dput* are read by *dget*.

EXAMPLES:

attach("/usr/abc/sdata") # attach another database (UNIX)

axes: (see title)

backsolve: Backsolve Upper- or Lower-triangular Equations.

backsolve(r,x,lower)

ARGUMENTS:

- r: upper or lower triangular matrix.
x: right-hand sides to equations.
lower: logical flag, default FALSE, to indicate lower-triangular r.

VALUE:

matrix like x of the solutions y to the equations $r.y=x$.

barplot: Bar Graph

barplot(height, width, names, angle, lines, col, space, inside, beside)

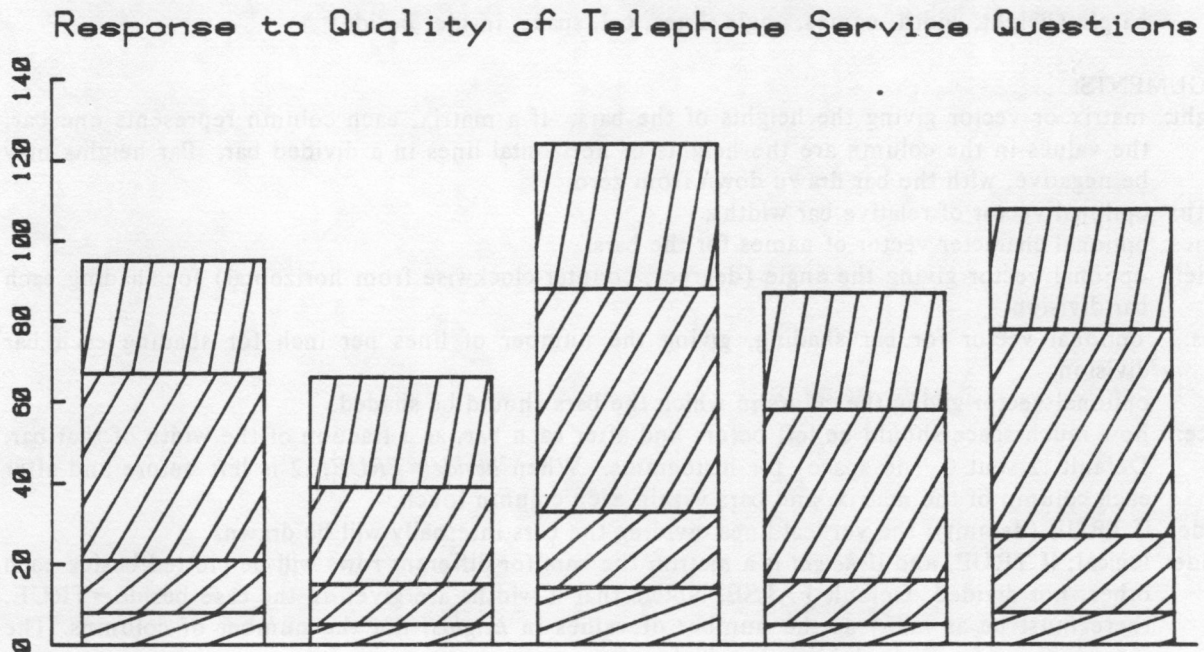
ARGUMENTS:

- height:** matrix or vector giving the heights of the bars. If a matrix, each column represents one bar; the values in the column are the heights of horizontal lines in a divided bar. Bar heights may be negative, with the bar drawn down from zero.
- width:** optional vector of relative bar widths.
- names:** optional character vector of names for the bars.
- angle:** optional vector giving the angle (degrees, counter-clockwise from horizontal) for shading each bar division.
- lines:** optional vector for bar shading, giving the number of lines per inch for shading each bar division.
- col:** optional vector giving the colors in which the bars should be shaded.
- space:** how much space should be left before and after each bar, as a fraction of the width of that bar. Default .2, but 0. (no space) for histograms. When *beside=TRUE*, .2 is left before and after each column of the matrix, and bars within each column touch.
- inside:** if TRUE (default), the vertical lines dividing the bars internally will be drawn.
- beside:** logical; if TRUE, and if *height* is a matrix, the bars for different rows will be plotted beside each other, not divided. Default FALSE. Notice that if widths are given in the case *beside=TRUE*, there must be as many as the number of values in *heights*, not the number of columns. The same is true of *names*, if this is supplied.

Graphical parameters may also be supplied as arguments to this function (see *par*).

EXAMPLES:

```
x←t(telsam.respons[1:5,])
x2←apply(x,2,"cumsum")
barplot(x2,angle=(1:4)*20)
title(main="Response to Quality of Telephone Service Questions",
sub="Divisions, Bottom to Top are Poor, Fair, Good, Excellent")
who←matrix("",4,5); who[3,]←encode("Int.",telsam.rowlab[1:5]) #1 label per col.
barplot(t(x),name=who,beside=TRUE,col=2:6) #same, with bars for each interviewer
```



Divisions, Bottom to Top, are Poor, Fair, Good, Excellent

BATCH: Batch (non-interactive) execution of S

S BATCH in out

ARGUMENTS:

- in: the name of a file containing S commands to be executed. The file may contain macro invocations, source and sink commands, etc. Also, deferred graphics may be carried out in batch mode. Interactive graphics devices (*tek14*, *hp72h*, etc.) produce no output when run in non-interactive mode.
- out: the name of a file which will receive all of the output from the run. By default, *out* will contain a listing of each expression in *in*, followed by any printed output produced by the expression.

Warning: execution begins immediately after *BATCH* is invoked. Processes running in batch are counted toward the total number of processes any one user is allowed. Running several versions of S (either batch or interactive) at the same time can cause you to exceed the process limit, and abort all of the runs.

pbeta qbeta rbeta: Beta Probability Distribution

```
pbeta(q, par1, par2)
qbeta(p, par1, par2)
rbeta(n, par1, par2)
```

ARGUMENTS:

q: vector of quantiles.
 p: vector of probabilities.
 n: sample size.
 par1: vector of shape parameters for numerator (>0).
 par2: vector of shape parameters for denominator (>0).

VALUE:

probability (quantile) vector corresponding to given quantile (probability) vector, or random sample (*rbeta*).

box: Add a Box Around a Plot

```
box(n)
```

ARGUMENTS:

n: number of times the box is drawn (i.e., heaviness of line). Default 1.

Graphical parameters may also be supplied as arguments to this function (see *par*).

boxplot: Box Plots

```
boxplot(arg1, arg2, ...,
        range=, width= varwidth=, notch=, names=, plot=)
```

ARGUMENTS:

argi: numeric structure or a structure containing a number of numeric components (e.g., the output of *split*). Missing values (NAs) are allowed.
 range=: controls the strategy for the whiskers and the detached points beyond the whiskers. By default, whiskers are drawn to the nearest value not beyond a standard range from the quartiles; points beyond are drawn individually. Giving *range=0* forces whiskers to the full data range. Any positive value of *range* multiplies the standard range by this amount. (The standard range is $1.5 \times (\text{inter-quartile range})$.)
 width=: vector of relative box widths. See also argument *varwidth*.
 varwidth=: logical flag. If TRUE, box widths will be proportional to the square-root of the number of observations for the box. Default FALSE.
 notch=: logical flag. If TRUE, notched boxes are drawn, where non-overlapping of notches of boxes indicates a difference at a rough 5% significance level. Default FALSE.
 names=: optional character vector of names for the groups. If omitted, names used in labelling the plot will be taken from the names of the arguments and from component names of structures.
 plot=: logical flag. If TRUE, the box plot will be produced. If false, only the calculated summaries of the arguments are returned. Default TRUE.

Graphical parameters may also be supplied as arguments to this function (see *par*).

VALUE:

if *plot* is FALSE, a structure with the components listed below. Otherwise function *bxp* is invoked with these components, plus optional *width*, *varwidth* and *notch*, to produce the plot.

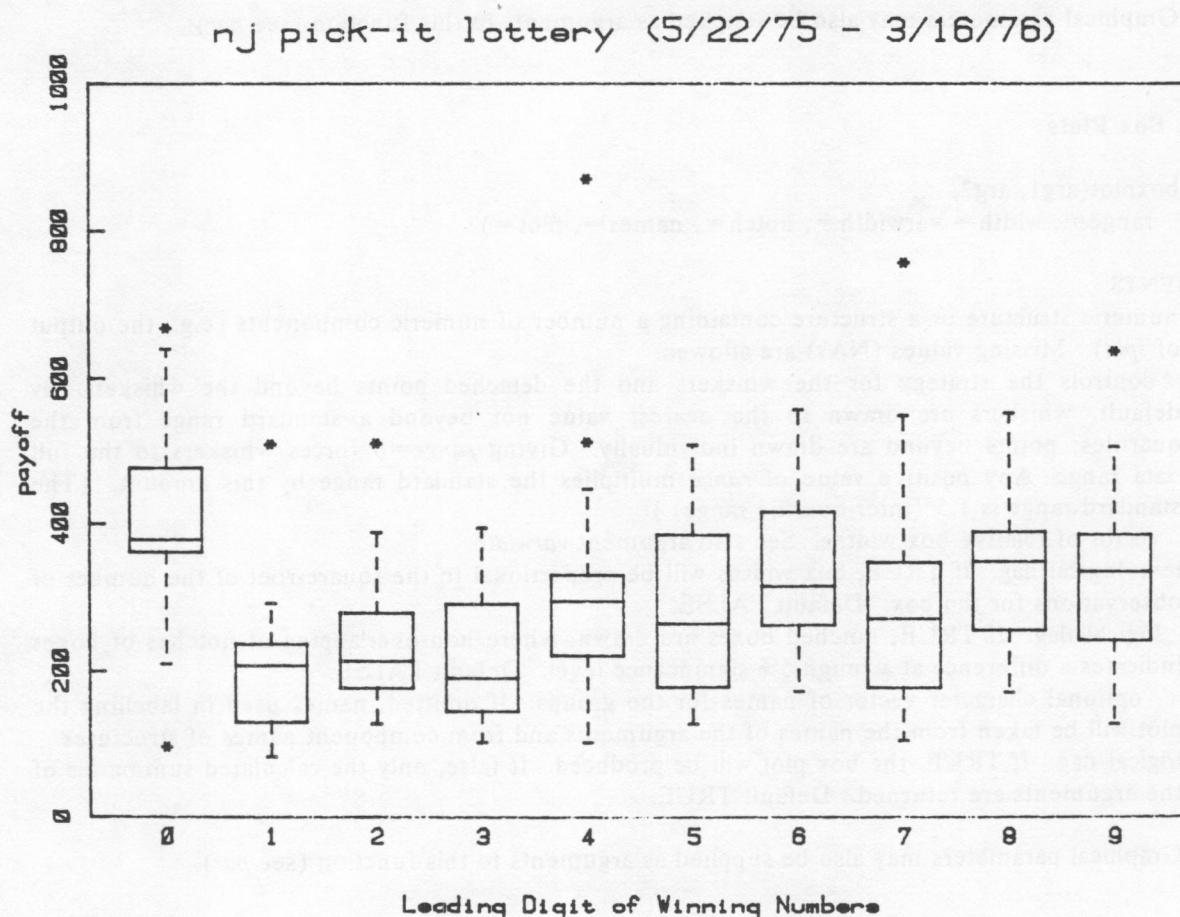
stats: matrix (5 by number of boxes) giving the upper extreme, upper quartile, median, lower quartile, and lower extreme for each box.
n: the number of observations in each group.
conf: matrix (2 by number of boxes) giving confidence limits for median.
out: optional vector of outlying points.
group: vector giving the box to which each point in *out* belongs
names: names for each box (see argument *names* above)

REFERENCE:

McGill, Tukey and Larsen, "Variations of Box Plots", *The American Statistician*, February 1978, Vol 32, No 1, pp 12-16.

EXAMPLES:

```
boxplot(split(salary,age),varwidth=TRUE,notch=TRUE)
boxplot(group1,group2,group3)
# the example plot is produced by:
boxplot(split(lottery.payoff,lottery.number%/100))
title(main=lottery.label,sub="Leading Digit of Winning Numbers",
ylab="payoff")
```



BPLOT: Submit Jobs for Batch Plotting

S BPLOT file device options

ARGUMENTS:

file: optional name of graphics save file, default *sgraph*.

device: optional name of batch graphics device. Can be *stare* for stare3 plots (the default), *photo* for FR80 microfilm photographs, *prism* for color plots.

options: flags which specify the service grade of the batch job. *-p* or *-1* is priority grade, *-s* or *-2* is standard grade (the default), *-e* or *-3* is economy grade, *-d* or *-4* is deferred grade.

If *file* is *sgraph*, the file is renamed before the batch job is submitted. This prevents S from appending new deferred graphics information to this file. When the file is renamed, its new name is printed. The file can be removed upon completion of plotting, resubmitted if the plotting does not complete because of batch computer problems, or resubmitted to another device (e.g., *photo*) for high-quality plots.

This command can be used on computers with direct access to batch plotting devices. See your local system administrator if in doubt.

EXAMPLES:

```
S BPLOT # submits file sgraph to stare
```

```
S BPLOT defer.12345 photo -3
```

```
# submits file defer.12345 to FR80 at economy grade
```

bxp: Boxplots From Processed Data

bxp(z, width, varwidth, notch)

ARGUMENTS:

z: structure whose components define the boxplot statistics, normally the result of a call to *boxplot* (which see for the components of *z*), but can be built up in any other way.

width: optional vector of box widths.

varwidth: logical flag, if TRUE, variable width boxes are drawn based on the *n* component of *z*. Default FALSE.

notch: logical flag, if TRUE, use the *conf* component of *z* to produce notched boxes. Default FALSE.

Graphical parameters may also be supplied as arguments to this function (see *par*).

c: Combine Values

`c(arg1, arg2, ...)`

ARGUMENTS:

argi: vector or structure. Missing values (NAs) are allowed.

VALUE:

vector which is the combination of all values from all arguments to the function. This includes all vectors that make up a structure. Thus *c* performs a function that is the opposite of *split*. Arguments that are NULL do not contribute anything to the result. See the last example.

EXAMPLES:

```
c(1:10,1:5,1:10)
c(1,2,3,5,7,11,13)
c(states,"Washington DC")
z←split(data,group) # split data by group then ..
c(sapply(z,"sort")) #sort components, combine result
x←NULL; for(i in seq(10))x←c(x,fun(i))
```

C: (see **PROGRAM**)

cancor: Canonical Correlation Analysis

`cancor(x, y)`

ARGUMENTS:

x,y: two matrices of data. The number of rows (number of observations) must be the same in each.

VALUE:

structure representing the canonical correlation analysis; i.e., a set of pairs of linear combinations of the variables in *x* and in *y* such that the first pair has the largest possible correlation, the second pair has the largest correlation among variables uncorrelated with the first pair, etc.

d: the correlations between the pairs of variables.

vx: the matrix of linear combinations of the columns of *x*. The first column of *vx* is the linear combination of columns of *x* corresponding to the first canonical correlation, etc.

vy: the matrix of linear combinations of the columns of *y*. The first column of *vy* is the linear combination of columns of *y* corresponding to the first canonical correlation, etc.

pcauchy qcauchy rcauchy: Cauchy Probability Distribution

```
pcauchy(q, par1, par2)
qcauchy(p, par1, par2)
rcauchy(n, par1, par2)
```

ARGUMENTS:

q: vector of quantiles.
 p: vector of probabilities.
 par1: vector of location parameters. Default is 0.
 par2: vector of scale parameters (>0). Default is 1.
 n: sample size.

VALUE:

probability (quantile) vector corresponding to given quantile (probability) vector, or random sample (*rcauchy*).

EXAMPLES:

```
rcauchy(20,0,10) #sample of 20, loc. 0, scale 10
```

cbind rbind: Form Matrix from Columns or Rows

```
cbind(arg1, arg2, ...)
rbind(arg1, arg2, ...)
```

ARGUMENTS:

argi: vector or matrix. Missing values (NAs) are allowed.

VALUE:

matrix composed by adjoining columns (*cbind*) or rows (*rbind*).

If several arguments are matrices, they must contain the same number of rows (*cbind*) or columns (*rbind*). If all arguments are vectors, the result will contain as many rows or columns as the length of the longest vector. Shorter vectors will be repeated.

If arguments are time-series, no attempt is made to relate their time parameters. See *tsmatrix* for this case.

Arguments that are NULL are ignored. See the last example.

EXAMPLES:

```
cbind(1,xmatr) %*% regr$coef # add column of ones
# to matrix multiply with coef vector
rbind(matrix,newrow1,newrow2) # add 2 new rows
x=NULL; for(i in seq(5)){... #compute z
x=cbind(x,z) }
```

ceiling floor trunc: Integer Values

ceiling(x)
 floor(x)
 trunc(x)

ARGUMENTS:

x: numeric structure. Missing values (NAs) are allowed.

VALUE:

structure of same mode as *x*. If *x* is an integer vector, the functions have no effect and *x* is returned. *trunc* truncates fractions toward zero, e.g., *trunc(1.5)* is 1., and *trunc(-1.5)* is -1. *floor* is largest integer $\leq x$. *ceiling* is smallest integer $\geq x$.

chapter: Include a User Chapter of Functions

chapter(file, detach)

ARGUMENTS:

file: character string, the name of the user chapter to be searched for S functions (and for their documentation). The most recently specified chapter is searched first (relevant in the case that two chapters have a function of the same name). Default is the user's own chapter (".").

detach: if TRUE, the named chapter is detached, rather than attached. Default FALSE.

EXAMPLES:

chapter("/usr/abc/testfuns")

CHAPTER: Utility Commands to Maintain a Chapter of S Functions**S CHAPTER**

S FUNCTION [options] name files ...

S MAKE name

A user's chapter of S functions is initialized (once) under the current directory by executing the command *CHAPTER*. Each time a new S function is to be written, the *FUNCTION* command is used (once). Whenever the function has been changed (either the interface routine or any of the other files on which the function depends), the *MAKE* command is used.

ARGUMENTS:

options: options controlling the interface to the function: a concatenation of any of the following: *-g* for a graphics function; *-r* for a function which needs to read the system common blocks (for example, because it does printing on OUTFC or generates random numbers) *-w* for a function which modifies the system parameters, and so must write the system common blocks (strongly discouraged for user functions), and *-d* for a device-driver function.

name: the name of the S function. It is assumed that there is a corresponding interface routine on a file *name.i* in the current directory.

files: names of files on which the function depends (in addition to standard S libraries), typically source files (which must end in ".r", ".f" or ".c") or library files. The *MAKE* command will try to keep up to date with source files, and will simply include all other files in the command that loads the function.

It is the user's responsibility to keep up to date any libraries, object files, etc. that are not given to *FUNCTION* as source files. If for some reason, the function must be reloaded even though none of the files on which it depends has been modified, precede the *MAKE* command by the command *rm name*.

If the set of files on which the function depends changes; e.g., the function now depends on a new ".r" file, the correct *FUNCTION* command should be given. This is the only time that *FUNCTION* must be re-invoked.

pchisq qchisq rchisq: Chi-Square Distribution

```
pchisq(q, parl)
qchisq(p, parl)
rchisq(n, parl)
```

ARGUMENTS:

q: vector of quantiles.
 p: vector of probabilities.
 parl: degrees of freedom (>0).
 n: sample size.

VALUE:

probability (quantile) vector corresponding to given quantile (probability) vector, or random sample (*rchisq*).

chol: Triangular Decomposition of Symmetric, Positive Definite Matrix

```
chol(x)
```

ARGUMENTS:

x: symmetric, positive definite matrix (e.g., correlation matrix or cross-product matrix).

VALUE:

upper-triangular matrix, *y*, such that $t(y) \%* y == x$.

chull: Convex Hull of a Planar Set of Points

```
chull(x, y, peel, maxpeel, onbdy, tol)
```

ARGUMENTS:

x,y: the co-ordinates of a set of points. A structure containing components *x* and *y* may also be given for *x*.
 peel: should successive convex hulls be peeled from the remaining points, generating a nested set of hulls? Default FALSE.
 maxpeel: maximum number of hulls that should be peeled from the data, default is to peel until all points are assigned to a hull. If *maxpeel* is given, it implies *peel*=TRUE.
 onbdy: should points on the boundary of a convex hull (but not vertices of the hull) be included in the hull? Default FALSE. If *peel* is TRUE, *onbdy* is also TRUE.
 tol: tolerance for determining inclusion on hull, default .0001.

VALUE:

- vector giving the indices of the points on the hull. If *peel* is TRUE, returns a data structure with components *depth*, *hull* and *count*.
- depth*: a vector which assigns a depth to each point, i.e., the number of hulls surrounding the point. Outliers will have small values of *depth*, interior points relatively large values.
- hull*: vector giving indices of the points on the hull. Along with *count*, determines all of the hull peels. The first *count[1]* values of *hull* determine the outermost hull, the next *count[2]* are the second hull, etc.
- count*: counts of the number of points on successive hulls.

EXAMPLES:

```
hull←chull(x,y)
plot(x,y)
hatch(x[hull],y[hull],space=0) # draw hull

p←chull(x,y,peel=T) # all hulls
which←rep(seq(p$count),p$count) # which peel each pt belongs to
s←split(p$hull,which)
for(i in seq(ncomp(s))) { # plot all peels
  j←s[i] # indices of points on ith peel
  hatch(x[j],y[j],space=0,lty=i)
}
```

code: Create Category from Discrete Data

```
code(x, level, label)
```

ARGUMENTS:

- x*: data vector. Missing values (NAs) are allowed.
- level*: optional vector of levels that the category should have. Any data value that does not match a value in *level* is coded as NA. Missing values (NAs) are allowed. Default is the set of unique values in *x*, sorted in ascending order.
- label*: optional vector of values to use as labels for the levels of the category. By default, the vector of levels is used.

VALUE:

- category structure with components *Data* and *Label*.
- Data*: integer vector indicating which level the category took on for each data value. NAs are returned where a data value in *x* did not match any value in *level*.
- Label*: character names for the levels of the category. If the argument *label* (either given or default) was not of mode CHAR, it will have been encoded.

EXAMPLES:

```
code(occupation) # "doctor", "lawyer", etc.
code(occupation,level=c("d","l"),label=c("Doctor","Lawyer")) # make readable labels
code(color,c("red","blue","green")) # only interested in 3 levels of color
```


coerce %m: Change Mode

```
coerce(x, mode)
x %m mode      #special operator
```

ARGUMENTS:

x: vector structure. Missing values (NAs) are allowed.
mode: desired mode for data: LGL, INT, REAL or CHAR.

VALUE:

vector whose data elements have the new mode.

EXAMPLES:

```
(0:40) %m CHAR    # convert 0 1 2 ... 40 to characters
runif(100)*10 %m INT # convert to integers (truncate)
```

col row: Matrix Column and Row Numbers

```
col(x)
row(x)
```

ARGUMENTS:

x: matrix. Missing values (NAs) are allowed.

VALUE:

integer matrix, containing the row number or column number of each element. If $z \leftarrow \text{row}(x)$, $z[i,j]$ is i ; if $z \leftarrow \text{col}(x)$, $z[i,j]$ is j .

See also *diag* for diagonal of matrix.

EXAMPLES:

```
x[row(x)>col(x)] # get strict lower triangle of x
x[row(x)-col(x)==1] # first sub-diagonal
```

compname: Component Names

```
compname(x)
```

ARGUMENTS:

x: hierarchical structure

VALUE:

character vector giving the names of the components of x .

Time-series have components named *Tsp* and *Data*. Matrices and arrays have components named *Dim* and *Data*.

EXAMPLES:

```
labels ← compname(struct)
compname(y) # find out structure of y without
```

```
#looking at all the data
```

contour: Contour Plotting

```
contour(x, y, z, v, nint=, add=, labex=)
```

ARGUMENTS:

- x:** vector containing x coordinates of grid over which z is evaluated.
y: vector of grid y coordinates.
z: matrix *len(x)* by *len(y)* giving surface height at grid points, i.e., *z[i,j]* is evaluated at *x[i]*, *y[j]*. The rows of *z* are indexed by *x*, and the columns by *y*. Missing values (NAs) are allowed.
v: vector of heights of contour lines. By default, approximately *nint* lines are drawn which cover most of the range of *z*. See the function *pretty*.
nint=: the approximate number of contour intervals desired, default 5. Not needed if *v* is specified.
add=: flag which if TRUE causes contour lines to be added to the current plot. Useful for adding contours with different *labex* parameter, line type, etc. Default FALSE.
labex=: the desired size of the labels on contour lines, default is same as standard (*cex*) character size. If *labex* is 0, no labels are used.

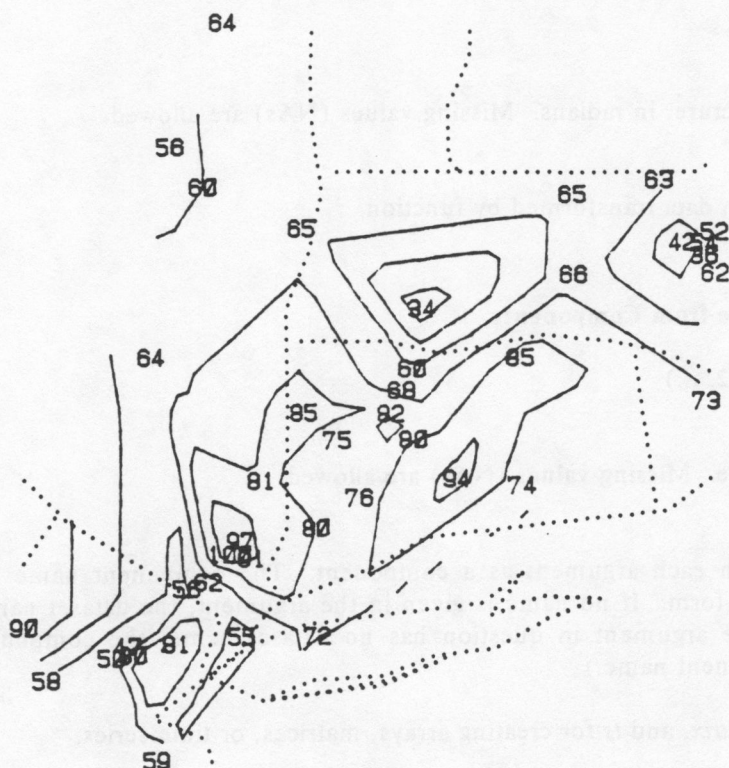
Graphical parameters may also be supplied as arguments to this function (see *par*).

The first argument may be a structure containing components named *x*, *y*, and *z*.

EXAMPLES:

```
rx←range(ozone.xy$x)
ry←range(ozone.xy$y)
usa(xlim=rx,ylim=ry,lty=2,col=2)
i←interp(ozone.xy$x,ozone.xy$y,ozone.median)
contour(i,add=T,labex=0)
text(ozone.xy,ozone.median)
title(main="Median Ozone Concentrations in the North East")
```


Median Ozone Concentrations in North East



cor var: Correlation and Variance

cor(x, y, trim)
var(x, y, trim)

ARGUMENTS:

x: matrix or vector.
y: matrix or vector. If omitted, same as x.
trim: the proportion trimmed in the internal calculations. Default 0.

VALUE:

correlations or variances (covariances), optionally with trimming. If x or y is a matrix, the result is a correlation or covariance matrix; the $[i,j]$ element corresponds to the i th column of x and the j th column of y.

Trimmed correlations are computed by the standardized sums and differences method.

Trimmed covariances are not yet implemented.

cos sin: Trigonometric Functions

```
sin(x)
cos(x)
```

ARGUMENTS:

x: numeric structure, in radians. Missing values (NAs) are allowed.

VALUE:

structure with data transformed by function.

cstr: Create Structure from Components

```
cstr(arg1, arg2, ...)
```

ARGUMENTS:

argi: arbitrary value. Missing values (NAs) are allowed.

VALUE:

structure with each argument as a component. The component name is the name given in `name=value` form. If no name is given in the argument, the dataset name of the argument is used. (If the argument in question has no dataset name, the component generated has an empty component name.)

See *array*, *matrix*, and *ts* for creating arrays, matrices, or time series.

EXAMPLES:

```
cstr(Data=read("gnp"),comment="gross national product")
cstr(a,d=b,1:50) #components named a, d, and no name
```

cumsum: Cumulative Sums

```
cumsum(x)
```

ARGUMENTS:

x: numeric structure.

VALUE:

structure like *x* where the *i*th value is the sum of the first *i* values in *x*.

cut: Create Category by Cutting Continuous Data

```
cut(x, breaks, labels)
```

ARGUMENTS:

x: data vector.

breaks: either a vector of breakpoints, or the number of equal-width intervals into which the data in *x* should be cut. If a vector of breakpoints is given, the category will have *len(breaks)-1* groups, corresponding to data in the intervals between successive values in *breaks* (after *breaks* is sorted). Data less than or equal to the first breakpoint or greater than the last breakpoint is returned as NA.

labels: character vector of labels for the intervals. Default is to encode the breakpoints to make up interval names (if *break* is a vector), or to use the names *Range 1*, etc., if *breaks* is a single integer.

VALUE:

a category structure with components *Label* and *Data*.

Data: vector of integers telling which group each point in *x* belonged to.

Label: vector of character names for each group.

EXAMPLES:

```
cut(x,3) # cut into 3 groups
cut(x, breaks) # cut based on given breakpoints
cut(x, pretty(x)) # approx 5 "pretty" intervals
```

cutree: Create Groups from Hierarchical Clustering

```
cutree(tree, k, h)
```

ARGUMENTS:

tree: hierarchical clustering tree structure, typically the output of *hclust*.

k: optional, the desired number of groups.

h: optional, the height at which to cut *tree* in order to produce the groups. Groups will be defined by the structure of the tree above the cut.

Exactly one of *k* and *h* must be supplied.

VALUE:

a vector with as many elements as the individual nodes making up the tree. The *i*th element of the vector gives the group number that individual *i* is assigned to. Individuals not in the current tree (if *tree* was a subtree from a larger original problem) are assigned group 0.

EXAMPLES:

```
cutree(tr←hclust(dist),k=5) #produce 5 groups
```

cycle: (see *time*)

defer: Control Deferred Graphics

defer(on, ask)

ARGUMENTS:

on: logical which enables (TRUE) or disables (FALSE) deferred graphics. Default TRUE.
ask: logical flag, if TRUE the user is asked "Save last frame?" before each new frame. Answers beginning with "N" or "n" cause the previous frame to be removed from the graphics save file. Default, FALSE, all frames are saved without user interaction.

All deferred graphics output goes to the file *sgraph*. If *sgraph* already exists, new material will be appended.

The graphics save file may be replotted on devices other than the device in effect during the *defer* function. This includes batch plotting, if available.

The graphics save file is in character format, and contains commands with the names *points*, *text*, *lines*, *segs*, and *eject* followed by the appropriate data values. The names *par* and *diff* reflect the graphical parameter changes made during plotting.

EXAMPLES:

```
defer # turn on deferred graphics
defer(FALSE) # turn off deferred graphics
defer(ask=TRUE) # turn on, ask about saving each frame
```

define: Define a Macro

define(file, pos, print)

ARGUMENTS:

file: optional character string giving the name of a file which contains definitions for one or more macros. If file is omitted, definitions are read from the terminal.
pos: database position on which macros are to be saved. Default 2.
print: logical, should names of defined macros be printed? Default TRUE for macros defined from file, FALSE for macros defined from the terminal.

define is used to create new S macro definitions. When *define* reads from terminal, it initially prompts with the string "N>", indicating that it expects the line giving the macro name and arguments. At this point, the user should give a line of the form

MACRO name(arg1/default value/,arg2/default/,...)

The prompt then turns to "D>", indicating a request for the definition of the macro. The definition continues until a line that just contains "END".

Define then prompts "N>" for another macro name. An empty line in response to this prompt terminates the definition process. Definitions from a file are terminated by the end of file.

See also *mprint* and *medit* to print and edit macro definitions.

The keywords *MACRO* and *END* must appear in upper case.

EXAMPLES:

```
> define      #define a macro incr
N>MACRO incr(x,y/1/)
D>x+y
D>END
N>
>
```

density: Estimate Probability Density Function

density(x, n, window, width, from, to)

ARGUMENTS:

x: vector of observations from distribution whose density is to be estimated.
 n: the number of equally spaced points at which to estimate density. Default 50.
 window: character string giving the type of window used in computation "cosine", "gaussian", "rectangular", "triangular". Default is "g" (one character is sufficient)
 width: width of the window. Default is width of histogram bar constructed by Doane's rule. The standard error of a Gaussian window is $width/4$.
 from:
 to: the n estimated values of density are equally spaced between *from* and *to*. Default is range of data extended by $width*3/4$ for gaussian window or $width/2$ for other windows.

VALUE:

plotting structure with two components, x and y.
 x: vector of n points at which density is estimated.
 y: density estimate at each x point.

REFERENCE:

Wegman, E. J. (1972). "Nonparametric probability density estimation", *Technometrics*, vol 14, 533-546.

EXAMPLES:

```
plot(density(x),type="b")
```

detach: Detach a Database from Search List

detach(file, pos)

ARGUMENTS:

file: optional character string giving the name of the database to be detached.
 pos: optional position (instead of name) of database in search list. If both arguments are omitted, the last database on search list is detached.

EXAMPLES:

```
detach("abc")
detach(pos=3) #detach shared database
```

devices: List of Supported Graphical Devices

tek14: Tektronix 4014 scope

tek14q: Tektronix 4014 scope with lower resolution, higher speed

tek12: Tektronix 4012 scope

tek10: Tektronix 4010, 4006 scopes (upper case only)

tek46h: Tektronix 4662 pen plotter, 8.5x11 paper (horizontal)

tek46v: Tektronix 4662 pen plotter, 8.5x11 paper (vertical)

tek46: Tektronix 4662 pen plotter, 11x17 paper

hp72h: Hewlett-Packard 7221 pen plotter, 8.5x11 paper (horizontal)

hp72v: Hewlett-Packard 7221 pen plotter, 8.5x11 paper (vertical)

hp72: Hewlett-Packard 7221 pen plotter, 11x17 paper

hp7220h

hp7220v Similar to *hp72h*, etc. for the Hewlett-Packard 7220 series.

hpgl Hewlett-Packard HP-GL plotters.

printer: Any printing terminal

stare: Stare3 electrostatic plotter, deferred plotting only.

prism: Prism color plotter, deferred plotting only.

photo: FR80 microfilm recorder photographic plot, deferred plotting only.

dget: Retrieve a General Structure from File

dget(file)

ARGUMENTS:

file: character string giving file name to read from. Default is the standard input.

VALUE:

data structure corresponding to contents of *file*.

Normally, the file was created through the function *dput*, possibly so that the data could be transmitted from another user and/or machine. The form of the data is a list notation:

(name mode length value1 value2 ...)

where *name* is a character string, and there are *length* values. In the case of a vector (*mode* being REAL, INT LGL or CHAR), the values are data of the corresponding mode. In the case of a hierarchical structure *valuei* is itself a list of the same form. In this case, the *mode* may be followed by the structure name.

Since the file is readable, one may edit the data on the file, although keeping the file in legal form may not be trivial.

EXAMPLES:

```
dget("copydata")
```

diag: Diagonal Matrices

```
diag(x, nrow, ncol)
```

ARGUMENTS:

x: matrix or vector.
nrow: optional number of rows of output matrix.
ncol: optional number of columns of output matrix.

VALUE:

the vector of diagonal elements of *x*, if *x* is a matrix. Otherwise, a matrix with *x* on its diagonal and zeroes elsewhere. If *x* is a scalar (vector of length 1), the value is an *x* by *x* identity matrix. By default, the matrix is square with zeros off the diagonal, but it can be made rectangular by specifying *nrow* and *ncol*.

EXAMPLES:

```
diag(xmat)    #extract diagonal
diag(diag(xmat)) # matrix of just diagonal of xmat
```

diary: Keep Diary of S Commands

```
diary(on, file)
```

ARGUMENTS:

on: if TRUE(the default), diary-keeping is turned on.
file: optional file name for diary entries, default is file "diary".

Each line that the user types while diary-keeping is active is entered into the diary file. Each line that comes from a source file or from a macro is also entered, predated by the comment character and indented to tell the level of source file nesting. Thus, the diary file can be used as a source file for re-execution of previously entered lines.

Each time the diary keeping is turned on, a comment line is entered into the diary file giving the current date and time. Also, note that user-typed comments are entered into the diary file.

EXAMPLES:

```
diary    # turn on diary file
diary(FALSE) # turn off diary file
```

diff: Create a Differenced Series

`diff(x, k, n)`

ARGUMENTS:

- `x`: a time series or vector. Missing values (NAs) are allowed.
- `k`: the lag of the difference to be computed (default=1).
- `n`: the number of differences to be done (default=1).

VALUE:

a time series which is the n -th difference of lag k for x .

EXAMPLES:

`d2x ← diff(x,1,2) # second difference of lag 1`

The start date of `d2x` will be 2 periods later than `x` due to the differencing.

discr: Discriminant Analysis

`discr(x, k)`

ARGUMENTS:

- `x`: matrix of data
- `k`: either the number of groups (if groups are equal in size), or the vector of group sizes; i.e., first $k[1]$ rows form group 1, next $k[2]$ group2, etc.

Note that the rows of `x` must be ordered by groups. See the system macro `?discr` for the use of a grouping vector to drive `discr`.

VALUE:

a structure describing the discriminant analysis, with the following components:

- `d`: vector of discriminant correlations (cor. between linear combination of variables and comb. of groups)
- `groups`: matrix of linear combinations of groups predicted.
- `vars`: matrix of linear combinations of variables.

Columns of `vars` give discriminant variables; i.e., `x %*% vars` produces the matrix of discriminant variables.

dist: Distance Matrix Calculation

`dist(x, metric)`

ARGUMENTS:

- x:** matrix (typically a data matrix). The distances computed will be among the rows of *x*. Missing values (NAs) are allowed.
- metric:** character string specifying the distance metric to be used. The currently available options are "euclidean" (the default), "maximum", "manhattan", and "binary". Euclidean distances are root sum-of-squares of differences, maximum is the maximum difference, manhattan is the sum of absolute differences, and binary is the proportion of non-zeroes that two vectors have in common.

VALUE:

- the distances among the rows of *x*. Since this structure can be very large, and since the result of *dist* is typically an argument to *hclust*, a special structure is returned, rather than a matrix.
- size:** the number of objects (that is, rows of *x*).
- data:** the $size*(size-1)/2$ values.

Missing values in a row of *x* are not included in any distances involving that row. Such distances are then inflated to account for the missing values. If all values for a particular distance are excluded by this rule, the distance is NA.

EXAMPLES:

```
dist(x,"max") # distances among rows by maximum
dist(t(x)) # distances among cols by euclidean
```

dprompt: Create Shell of Documentation for Datasets

`dprompt(name, list, file)`

ARGUMENTS:

- name:** common name for datasets being documented.
- list:** optional list of names of datasets to be documented together. May be NULL.
- file:** optional file name for documentation file. Default is *name.d*.

This function is ordinarily used via the system macro "?prompt".

dput: Save a Dataset on a File

`dput (x, file)`

ARGUMENTS:

- x:** the data structure to be saved.
- file:** character string giving the file name where the data structure is to be stored. Default is the user's terminal.

VALUE:

the dataset *x*.

The function *dget* can re-create the dataset from *file*. See also *dump* to write several datasets to

a file.

EXAMPLES:

```
dput(abc,"oldabc") # store data structure abc on file oldabc
def=dget("oldabc") # read in data structure
```

dump: Dump Datasets to a File

```
dump(list, file, pos)
```

ARGUMENTS:

- list: character vector of the names of the S datasets to dump. Typically, this is the result of the *list* function.
- file: character name of file where datasets are to be dumped. Default "dumpdata".
- pos: position on search list from which datasets are to be taken. Default 0, use first dataset with the correct name encountered on search list.

The use of *dump* is usually to dump datasets for transmission to another machine, or for archive purposes. One useful technique is to dump all the data under a prefix; e.g., when you want to share the data on a specific project with someone on another machine.

The function *restore* is used to recreate datasets written by *dump*. The functions *dump* and *restore* are not affected by the current prefix.

EXAMPLES:

```
dump(list("longley.*"),"longleydump") #all longley.*
dump(list(pos=2),"dbdump",pos=2) #the WHOLE save database
# note need to use pos= to both functions
```

edit: Edit Dumped Expressions or Character Vectors

```
edit() #for editing the dumped expression
again() #to re-evaluate dumped expression
edit(x) #for character vector
```

ARGUMENTS:

- x: optional character vector to be edited. If *x* is omitted, *edit* operates on the last expression dumped because of a syntax or an execution error.

EFFECT:

If *x* is given, the edited data is assigned, with the same name, on the working database. Note that this implies that it is impossible to edit an expression (as opposed to a dataset), since this has no name for the assignment: *edit(encode(1:5))* will not work.

After *edit* is invoked, the data from the dumped expression or character vector is written to a hidden file, one element of the vector per line of the file. The *ed* text editor is invoked and the file is read.

At this point, *ed* commands can be used to modify the text. When a suitable version of the text is produced, execute the following two *ed* commands: *w* to write the edited text back to the file, and *q* to exit from *ed*.

After exiting from *ed*, the file is read by *edit*. If the data was a dumped expression, a hidden call to *source* causes the edited expression to be executed. Notice that this edited version stays around, for another call to *edit* to make further changes. If *x* was given, the edited expression is stored on the working database.

To suppress execution of the edited expression do ****not**** just quit without writing (this leaves the expression as before). Instead, use the *ed* command *l,\$s.*//*, then *w*; this leaves nothing to execute.

The function *again* is useful to re-execute expressions that were dumped because of execution errors that have been remedied, e.g., when a graphic device has been activated, when a missing dataset has been created, or when a prefix has been specified.

Use the function *medit* to edit macros.

EXAMPLES:

```
edit(state.name) #edit the vector of state names.
```

EDITDOC: (see PROMPT)

eigen: Eigen Analysis of Symmetric Matrix

```
eigen(x, n, large)
```

ARGUMENTS:

- x:** matrix to be decomposed. Must be square and symmetric.
- n:** number of eigenvalues and corresponding eigenvectors wanted from *x*. Default is to return all eigenvalues.
- large:** logical flag; if TRUE (the default), returns the *n* largest eigenvalues; otherwise, returns the *n* smallest eigenvalues.

VALUE:

- a structure containing the eigenvalues and eigenvectors.
- values:** vector of *n* eigenvalues; note that the values are always in ascending order, regardless of *largest*.
- vectors:** matrix with *nrow(x)* rows and *n* columns. Each column is the eigenvector corresponding to the eigenvalue which is the corresponding element of *values*.

EXAMPLES:

```
cors←cor(x,y,trim=.1)
pprcom←eigen(cors)
```

else: (see *syntax*)

encode: Encode Text and Numeric Data

```
encode(arg1, arg2, ..., sep=)
```

ARGUMENTS:

argi: vectors which may be either numeric, logical, or character. If the argument is character, its elements are concatenated with the next arguments as is. Otherwise, the values are encoded into character strings first.

sep=: the character string to be inserted between successive arguments. Can be "" for no space. Default single space (" ").

VALUE:

character vector, with length equal to the maximum of the lengths of the arguments. The i-th element of the result is the concatenation of the i-th elements of the arguments (encoded if not originally character strings). If the length of the argument is less than the maximum, elements of that argument are repeated cyclically. In particular, an argument can be a single element, to appear in each element of the result.

EXAMPLES:

```
encode("no.",1:10) # gives "no. 1", "no. 2" ...
encode(state.name,"pop=",pop) # "alabama pop= 12.345"...
```

end nper start: Time Series Parameters

```
start(x)
end(x)
nper(x)
```

ARGUMENTS:

x: time series. Missing values (NAs) are allowed.

VALUE:

vector giving the appropriate time parameter of *x*. In the case of *start* and *end*, the result has 2 values, the year and month.

The values of these functions are appropriate for any time-series function taking arguments *start*, *end*, *nper*.

EXAMPLES:

```
# assume ts is monthly from Jan. 1970 to Jul. 1975
start(ts) # is the vector c(1970,1)
end(ts)   # is the vector c(1975,7)
nper(ts)  # is 12
xyz=ts(data,start=start(gnp),nper=nper(gnp))
# create time series beginning at same time as gnp
```


exp log log10 sqrt: Math Functions

exp(x)
 log(x)
 log10(x)
 sqrt(x)

ARGUMENTS:

x: numeric structure. Missing values (NAs) are allowed.

VALUE:

structure like *x* with data transformed by the function. *exp* is exponential, *log* natural logarithm, *log10* common log, *sqrt* square root.

Argument values which cannot be handled (e.g., negative values for square root) evaluate to NA and generate a warning message.

extract: Extract Columns or Fields as S Datasets

?extract(file)

S extract file [desfile extfile] # outside of S

ARGUMENTS:

file: the name (unquoted) of a file containing input data, either in specific columns or else as fields separated by a field separator character.

desfile: the name of a description file specifying how to extract data from *file*. Defaults to *file.des*.

extfile: the name of the file onto which the external form of the extracted S data structures will be written. Defaults to *file.ext*.

The macro form extracts the datasets, writes them to the extract file and uses the *restore* function to bring these datasets into the user's working database. The utility version just forms the extract file, which could then be shipped to another computer system, etc.

The *description file* lists the name of each of the datasets to be extracted and their position in the records of the file. In *column* form of description, each line of the description file is of the form:

name mode start length

with *name* the name of the extracted dataset, *mode* the desired mode of the extracted data (R, I, C for REAL, INTEGER and CHARACTER), *start* the beginning column for the data items on each record, and *length* the number of columns in the item. In the *field* form of input, the first line of the description file should be of the form

-f%

where "%" is the desired field separator character (tab by default). Remaining lines of the description file are then of the form:

name mode field

with *name* and *mode* as before, and *field* defining which field of the record becomes the data item.

pf qf rf: f Probability Distribution

```
pf(q, par1, par2)
qf(p, par1, par2)
rf(p, par1, par2)
```

ARGUMENTS:

q: vector of quantiles.
p: vector of probabilities.
par1: vector of degrees of freedom for numerator.
par2: vector of degrees of freedom for denominator.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*rf*).

EXAMPLES:

```
rf(10,5,15) # sample of 10 with 5 over 15 df
```

faces: Plot Symbolic Faces

```
faces(x, which, labels, head, max, nrow, ncol, fill, scale, byrow)
```

ARGUMENTS:

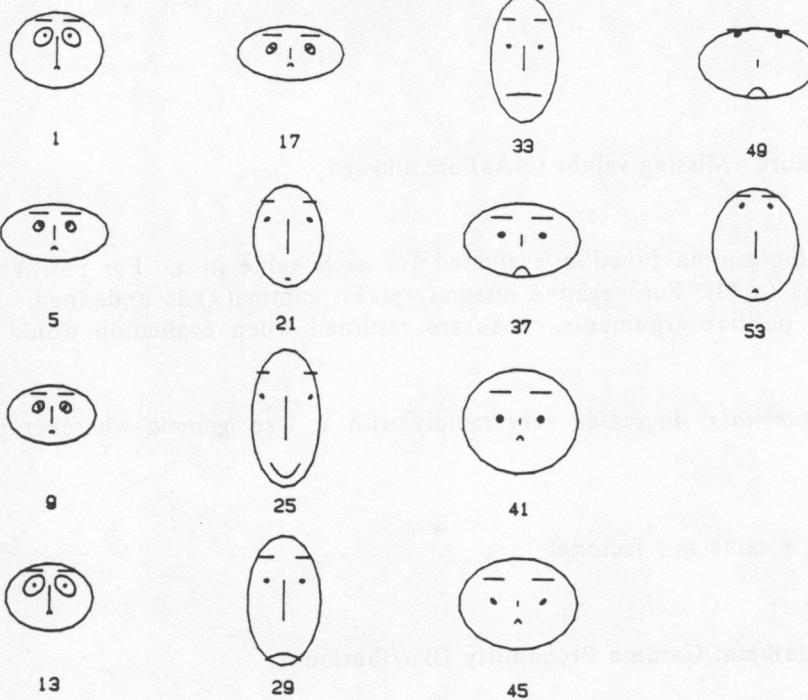
x: matrix of data values. Missing values (NAs) are allowed.
which: the columns of *x* to be used as the first, second, etc. parameter in the symbolic face. Default *1:min(15,ncol(x))*. See NOTE below for meaning of parameters.
labels: optional character vector of labels for the faces (i.e., for the rows of *x*).
head: optional character vector to use as the heading for the plot.
max: a suggested value for the number of rows and columns to go on each page. By default, all the faces will be fitted onto one page.
nrow:
ncol: optionally, may be given to specify exactly the number of rows and columns for the array of plots on each page.
fill: if TRUE (the default), all unused parameters of the face will be set to their nominal (midpoint) value. If FALSE, all features corresponding to unused parameters will not be plotted.
scale: if TRUE (the default), the columns of *x* will be independently scaled to (0,1). If FALSE no scaling will be done. The data values should then be scaled to the same overall range by some other means; e.g., by scaling the whole of *x* to the range (0,1).
byrow: if TRUE, plots produced in row-wise order; if FALSE (the default) plots are produced in column-wise order.

NOTE: the feature parameters are: 1-area of face; 2-shape of face; 3-length of nose; 4-loc. of mouth; 5-curve of smile; 6-width of mouth; 7,8,9,10,11-loc., separation, angle, shape and width of eyes; 12-loc. of pupil; 13,14,15-loc., angle and width of eyebrow.

EXAMPLES:

```
faces(chernoff2,head="Chernoff's Second Example")
```


Chernoff's Second Example



floor: (see ceiling)

for: (see syntax)

FORTTRAN: (see PROGRAM)

frame: Advance Graphics Device to Next Frame or Figure

frame

Upon receipt of this command, the graphics device will eject to a new page, clear the screen, or do whatever is appropriate for the device. (In multiple figure mode, will advance to the next figure).

FUNCTION: (see CHAPTER)

gamma lgamma: Gamma Function (and its Natural Logarithm)

```
gamma(x)
lgamma(x)
```

ARGUMENTS:

x: vector structure. Missing values (NAs) are allowed.

VALUE:

gamma or log-gamma function evaluated for each value in x. For positive integral values, gamma(x) is (x-1)!. For negative integral values, gamma(x) is undefined. Function *lgamma* allows only positive arguments. NAs are returned when evaluation would cause numerical problems.

Note that *gamma(x)* increases very rapidly with x. Use *lgamma* wherever possible to avoid overflow.

EXAMPLES:

```
gamma(6) # same as 5 factorial
```

pgamma qgamma rgamma: Gamma Probability Distribution

```
pgamma (q, parl)
qgamma (p, parl)
rgamma (n, parl)
```

ARGUMENTS:

q: vector of quantiles.
p: vector of probabilities.
parl: vector of shape parameters (>0).
n: sample size.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*rgamma*) from the gamma distribution.

EXAMPLES:

```
rgamma(20,10) # sample of 20 with shape parameter 10
```

get: Access Data on a Database

```
get(name1, name2, ...,pos=)
```

ARGUMENTS:

namei: the character strings which give the main name and the component names of the data; i.e., *name1* is the *name* of the structure and *name2*, *name3*, etc. are the names of the component, subcomponent, etc.
pos=: the position of the database from which the data is to be retrieved. If *pos==0*, all databases are searched for the dataset. Default 0.

VALUE:

the data found.

The function *get* is the explicit analogue to what happens automatically when the name of a dataset or a component appears in an expression. Occasionally *get* is used explicitly explicitly because the names can be constructed or read in, as in the example below.

EXAMPLES:

```
for(i in 1:10){
  name ← encode("y",i,sep="")
  plot(x,get(name)) }
#scatter plot (x,y1), (x,y2),...(x,y9)
```

glim: Interface to the GLIM system

```
glim(var1, var2, ..., y=, error=, den=, link=, weight=, offset=
      fit=, names=)
kill("Sglim")
```

The *glim* function provides the interface to the GLIM system, see below, for analysis of generalized linear models. *kill* removes the glim system. Invoke it once before quitting.

ARGUMENTS:

vari: the variates (in GLIM terminology) to be used in the fit. These should all be vectors (matrices) of the same length (number of rows). If no *y* argument is given, the last of the variates is taken to be the response. The name *vi* is given to the *i*th variate; columns of matrices count as variates.

y=: optionally, the data for the predicted (*y*) variable. By default, the last variate is assumed to be *y*.

error=:

den=:

link=:

weight=:

offset=:

fit=: these are respectively the optional values for the *\$error*, *\$den*, *\$link*, *\$off* and *\$fit* directives for GLIM. For those not familiar with GLIM, see below and in the GLIM Manual. If *fit* is missing, the default model will be " $v_1 + v_2 + \dots + v_n$ ", where *vi* is the *i*th variate name.

The *error*, *link* and *fit* arguments are character strings (lower case only). The other arguments are datasets, with the same number of observations as the variates.

names=: optional, vector of variate names to be used in the fit description. These must be unique in 4 characters, and be composed of lower case letters and digits (GLIM restrictions). If omitted, names "v1", "v2", ... will be used. The *y* variate is always named "y".

It is possible to fit many models to the same data. If a previous *glim* fit has defined variates, any call to *glim* that does not change the variates, *y*, *offset* or *weight* will fit a new model to the current data. In particular, GLIM accepts just changes to the model (see example).

VALUE:

a data structure describing the result of the GLIM fit. The arguments *error* through *fit* are included in the results, along with the following.

dv: the value of the deviance (roughly, negative log-likelihood) for the model.

coef: the coefficients of the fitted model. Note that in the case of overdetermined models (e.g., analysis of variance) the parameterization is that of GLIM, which is somewhat unconventional in that it makes the first effect zero, rather than the sum of the effects.

fv: the vector of fitted values from the model.

vc: the variance-covariances, returned as the lower-triangle of a matrix.

labels: the labels associated with the elements of *coef*, made up of the names of the variables or categories, plus level numbers for the categories.

names: the names of the variables passed to GLIM as predictors, response and optional weights and offset.

EXAMPLES:

```
glim(a,b,c,resp,error="p",weight=w1,fit="v1*v2+v3")
glim(fit="-v3") # the model is now "v1*v2"
```

REFERENCES:

Baker, R. J. and Nelder, J. A. (1978) *The GLIM System*. Numerical Algorithms Group, 7 Banbury Road, Oxford, ENGLAND.

Chambers, J. M. and Schilling, J. M. (1981). "S and GLIM: an experiment in interfacing statistical systems" *Bell Laboratories Memorandum*.

BACKGROUND:

For those unfamiliar with GLIM, the following brief description may get things started. For detailed documentation see the first reference.

GLIM fits linear models, in which the linear predictor may be transformed by the *link* function before being used to predict *y*. The predictor is the sum of linear terms plus an *error*, which is assumed to come from one of a standard set of distributions. If the predictor variates are categories (*factors* in GLIM terminology), the GLIM model will contain coefficients for all the levels of the factor (except the first; see *coef* above). If more than one factor appears in the model, one may choose to fit also the *interaction* terms for two or more factors.

The model used by GLIM is controlled by the *fit* directive. This is a character string containing the names of the factors, joined by "+", "*" and ".". These operators mean to include both the terms, both the terms plus the interactions and only the interaction, respectively.

The commonly useful GLIM models are: the analysis of variance (default *error* and *link*); loglinear models for contingency tables (*link*="l"); logit and probit models (*link*="g" and *link*="p"). Other values for the *link* argument are "r" (reciprocal), "c" (complementary log-log), "s" (square root), "i" (identity) and "e number" (exponent to *number*). The error may be "n" (normal), "p" (poisson), "b" (binomial) or "g" (gamma); when binomial errors are used, the denominator vector must be provided as *den*. See the GLIM manual for interpretation of all this, for examples and for the use of the *weight* and *offset* directives.

gs: Gram-Schmidt Decomposition

gs(x)

ARGUMENTS:

x: matrix to be decomposed.

VALUE:

orthogonal decomposition of the matrix $x = r \%* q$, using the Gram-Schmidt decomposition with iterative re-orthogonalization.

r: upper-triangular matrix, of order $ncol(x)$.

q: matrix of the same dimensions as x, whose columns are orthogonal and of unit euclidean norm.

hardcopy: On-line Copy of Current Display

hardcopy(delay)

ARGUMENTS:

delay: number of seconds to delay while copy is being made. Default 30.

hardcopy currently works with Tektronix 4000 series scopes. See *defer* and *BPLOT* for general deferred graphics.

hatch: Shade in a Polygonal Figure

hatch(x, y, space, angle, border)

ARGUMENTS:

x,y: coordinates of the vertices of a polygon, listed in order, i.e., the i th point is connected to the $i+1$ st. It is assumed the polygon closes by joining the last point to the first. A structure containing components x and y can also be given.

space: inches of space between adjacent shading lines. Default .1. If *space* is zero, no hatch lines will be drawn.

angle: angle of shading lines in degrees measured counterclockwise from horizontal. Default 45.

border: should border of polygonal region be drawn? Default TRUE.

Graphical parameters may also be supplied as arguments to this function (see *par*).

EXAMPLES:

hatch(rdpn()) # read pen positions, draw and shade polygon

hclust: Hierarchical Clustering

`hclust(dist, method, sim)`

ARGUMENTS:

- dist:** a distance matrix structure. Normally this will be the result of the function *dist*, but it can be any data of the form returned by *dist*, or a full, symmetric matrix.
- method:** character string giving the clustering method. The three methods currently implemented are "average", "connected" (single linkage) and "compact" (complete linkage). The default is "compact". (The first three characters of the method are sufficient.)
- sim=:** optional structure replacing *dist*, but giving similarities rather than distances. Exactly one of *sim* or *dist* must be given.

VALUE:

a "tree" representing the clustering, consisting of the following components:

- merge:** an (n-1) by 2 matrix, if there were n objects in the original data. Rows 1,2,...,n-1 of *merge* describe the merging of clusters at steps 1,2,...,n-1 of the clustering. If an element in the row is of the form -j, then object j was merged at this stage. If the element of *merge* is of the form +i, then the merge was with the cluster formed at the (earlier) stage i of the algorithm.
- height:** the clustering "height"; that is, the distance between clusters merged at the successive stages.
- order:** a vector giving a permutation of the original objects suitable for plotting, in the sense that a cluster plot using this ordering will not have crossings of the branches.

EXAMPLES:

```
hclust(dist(x))
hclust(dist(x),"ave")
```

help: On-line Documentation

```
help("name") # information about functions
help(macro="name") # information about macros
help(dataset="name") # information about datasets
```

ARGUMENTS:

- name:** character string, giving the name of a function, operator, macro, or dataset. If omitted, documentation on "help" is given (this documentation).

If information is found on *name*, the documentation will be printed. This is the same documentation as in the S manual.

EXAMPLES:

```
help("cstr") # documentation for function cstr
help("+") # documentation on addition (& other arithmetic)
help(macro="row") # ?row documentation
help(dataset="longley") # Longley dataset info
```


hist: Plot a Histogram

hist(x, nclass, breaks, density, plot, angle, lines, inside)

ARGUMENTS:

- x:** numeric vector of data for histogram.
nclass: optional recommendation for the number of classes the histogram should have. Default is such that $2^{**}nclass$ is of the order of $len(x)$.
breaks: optional vector of the break points for the histogram classes. The first value of breaks should be smaller than any data point; the last value of breaks should be larger than any data point. If omitted, evenly-spaced break points are determined from *nclass* and the extremes of the data.
density: logical flag. If TRUE, y axis will be on a density scale, otherwise y will be counts. Default FALSE.
plot: logical flag. If TRUE, the histogram will be plotted; if FALSE, the vectors *counts* and *breaks* will be returned. Default TRUE.
angle: optional angle of lines to be used for shading histogram (degrees, counterclockwise from the horizontal). Default: no shading.
lines: optional number of lines per inch for histogram shading.
inside: flag, TRUE (the default) if vertical lines should be drawn to separate the bars.

Graphical parameters may also be supplied as arguments to this function (see *par*).

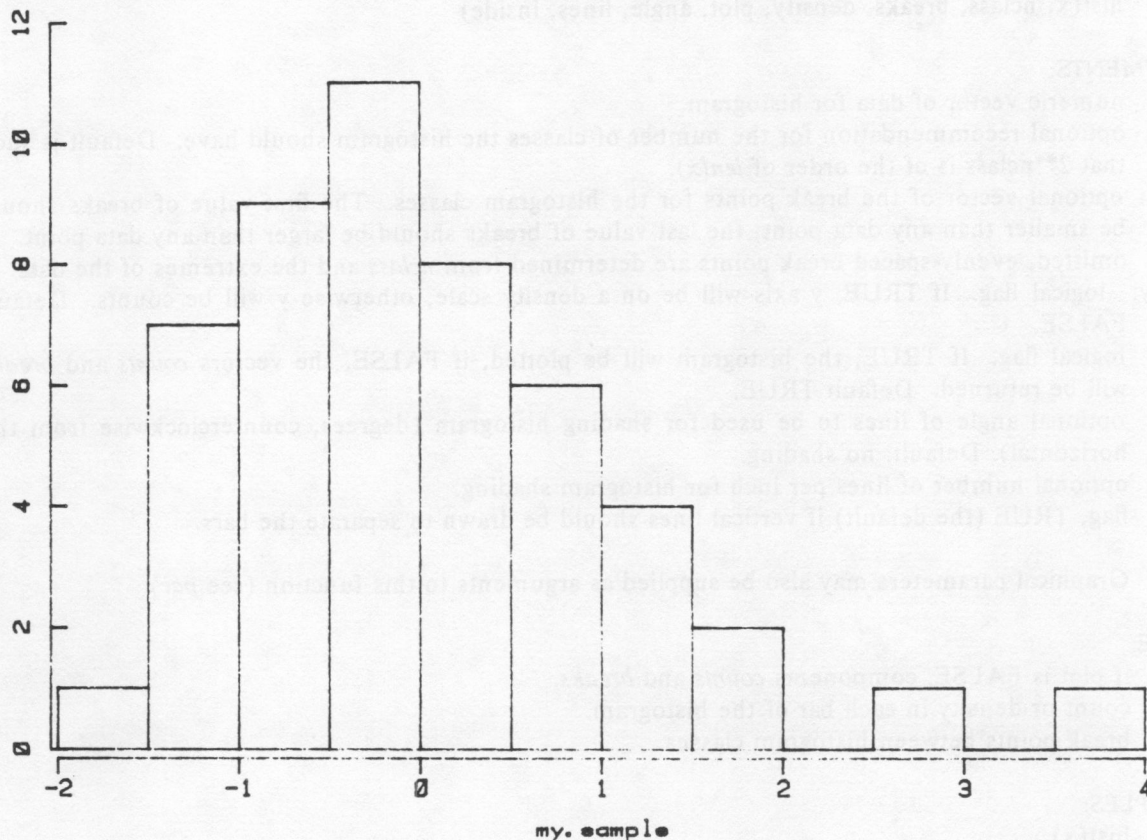
VALUE:

- if plot is FALSE, components *counts* and *breaks*.
counts: count or density in each bar of the histogram.
breaks: break points between histogram classes.

EXAMPLES:

```
hist(x)
hist(x,nclass=15)
# the example plot is produced by:
my.sample←rt(50,5)
lab—"50 samples from a t distribution with 5 d. f."
hist(my.sample,main=lab)
```

50 samples from a t distribution with 5 d.f.



hp72 hp72h hp72v: Hewlett-Packard 7221a Pen Plotter

```
hp72(width, height, ask, auto, with2621)
hp72h
hp72v
```

ARGUMENTS:

width: width of plotted surface in inches. Default 8 for *hp72v*, 10 for *hp72h*, 15 otherwise.
height: height of plotted surface in inches. Default 8 for *hp72h*, 10 otherwise.
ask: logical, should user be prompted by "GO?" prior to advancing to new frame? Default TRUE.
auto: logical, can device automatically advance the paper (H-P 7221S)? Default FALSE.
with2621: logical, is the plotter configured with H-P 2621 terminal? Default FALSE. If TRUE, special output is provided to make the plotter and terminal work together.

These commands identify Hewlett-Packard 7221 series plotters. The plotter may be used with standard 8.5 by 11 inch paper with the long dimension horizontally (*hp72h*) or vertically (*hp72v*), or with 11 by 17 paper (*hp72*). The arguments listed after *hp72* can also be given with *hp72h* or *hp72v*.

Whenever a new plot is about to be produced, the message "GO?" appears on the terminal. At this time, new paper may be loaded, pens changed, etc. When ready for plotting to proceed, simply hit carriage return.

When graphic input (*identify*, *rdpen*) is requested, the *enter* button will light. The pen should be

positioned by means of the 5 directional buttons. When the desired pen position is reached, depress the *enter* button. To terminate the input, hit carriage return on the terminal (the coordinates of the pen are not transmitted).

Character sizes may be changed to any desired value. Color can be changed to any of the values 1 to 4, indicating the pens in stables 1 to 4. Characters can be rotated to any orientation. Different line styles (1, 2, ...) are available, with patterns becoming more spread-out as the line style is larger.

ALWAYS BE SURE that the pen is back in the holder when you are done with the terminal. Either the S function *q* or hitting in succession the *enter* button followed by the button below the currently empty holder will do this. Users who fail in this respect will be devoured by the dry-pen dragon.

NOTE:

The switches on the back of the device should be set to reflect the correct parity settings for the computer system. Incorrect parity settings will cause failures of the digitizing functions, e.g., *rdpen* and *identify*.

A device must be specified before any graphics functions can be used.

hp7220h: (see **hpgl**)

hp7220v: (see **hpgl**)

hp7225: (see **hpgl**)

hpgl hp7220h hp7220v hp7225: Hewlett-Packard HP-GL Plotters

hpgl(width, height, ask, auto, color)

hp7220h

hp7220v

hp7225

ARGUMENTS:

width: width of plotted surface in inches. Default 8 for *hp7220v*, 10 otherwise.

height: height of plotted surface in inches. Default 10 for *hp7220v*, 8 otherwise.

ask: logical, should user be prompted by "GO?" prior to advancing to new frame? Default TRUE.

auto: logical, can device automatically advance the paper (H-P 7220S)? Default FALSE.

color: integer reflecting the degree of color-plotting support provided by the device. 0=color changes ignored, 1=prompt user when color changes (to allow manual pen changes, etc), 2=device has automatic color changing capability (*hp7220*).

These commands identify Hewlett-Packard plotters which accept the HP-GL instruction set. This includes pen plotter models 7220 and 7225. The plotters may be used with standard 8.5 by 11 inch paper with the long dimension horizontally (*hp7220h*) or vertically (*hp7220v*), or with other paper sizes by means of arguments *width* and *height*. The arguments listed after *hpgl* can also be given with *hp7220h*, *hp7220v* or *hp7225*.

Whenever a new plot is about to be produced, the message "GO?" appears on the terminal. At this time, new paper may be loaded, pens changed, etc. When ready for plotting to proceed, simply hit carriage return.

When graphic input (*identify*, *rdpen*) is requested, the *enter* button will light. The pen should be positioned by means of the 4 directional buttons. When the desired pen position is reached, depress the *enter* button. To terminate the input, hit carriage return on the terminal. The coordinates of the terminating point are not transmitted.

Character sizes may be changed to any desired value. Color can be changed to any of the values 1 to 4, indicating the pens in stables 1 to 4. Characters can be rotated to any orientation. Different line styles (1, 2, ...) are available, with patterns becoming more spread- out as the line style is larger.

ALWAYS BE SURE that the pen is back in the holder when you are done with the terminal. Either the S function *q* or hitting in succession the *enter* button followed by the button below the currently empty holder will do this.

NOTE:

The switches on the back of the device should be set to reflect the correct parity settings for the computer system. Incorrect parity settings will cause failures of the digitizing functions, e.g., *rdpen* and *identify*.

A device must be specified before any graphics functions can be used.

identify: Identify Points on Plot

`identify(x, y, label, n, plot, atpen, offset)`

ARGUMENTS:

- x,y:** co-ordinates of points that may be identified. A time series or structure containing *x* and *y* may also be given in place of *x,y*. Missing values (NAs) are allowed.
- label:** optional character vector giving label for each of the points. If supplied, must have the same length as *x* and *y*. Default label is the point number.
- n:** maximum number of points to identify.
- plot:** flag, if TRUE (the default), *identify* plots the labels of the points identified. In any case, the subscripts are returned.
- atpen:** flag, if TRUE (the default) plotted identification is relative to pen position when point is identified; otherwise, plotting is relative to the identified *x,y* value. Useful for controlling position of label when points are crowded.
- offset:** identification is plotted as a text string, moved *offset* character widths from the point (default 1). If the pen was left (right) of the nearest point, the label will be offset to the left (right) of the point.

Graphical parameters may also be supplied as arguments to this function (see *par*).

VALUE:

indices (in *x* and *y*) corresponding to identified points.

See graphics device command for details on graphical input techniques.

The nearest point to pen position is identified, but must be at most half-inch from pen. In case of ties, earliest point is identified.

EXAMPLES:

```
bad ← identify(x,y,plot=FALSE)
xgood ← x[-bad]      # eliminate identified "bad" points
ygood ← y[-bad]      # from x and y
```

if: (see **syntax**)

ifelse: Conditional Data Selection

```
ifelse(cond, true, false)
```

ARGUMENTS:

cond: logical structure.
true: vector containing values to be returned if *cond* is TRUE.
false: vector containing values to be returned if *cond* is FALSE.

VALUE:

vector like *cond*. The result is made up element-by-element from the values from *true* or *false* depending on *cond*. If *true* or *false* are not as large as *cond*, they will be repeated cyclically. Missing values (NAs) are allowed. NA values in *cond* cause NA to be returned.

EXAMPLES:

```
ifelse(x>1,1,x) # gives the value 1 if x>1 else gives x
               # equivalent to pmin(1,x)
```

Evaluation of arguments is done before execution of *ifelse*. Divide-by-zero will already have occurred in *ifelse(x==0,NA,1/x)* by the time *ifelse* is executed. Avoid the zero-divide by *1/ifelse(x==0,NA,x)*.

The *if* syntax construct of the S language provides conditional evaluation.

interp: Bivariate Interpolation for Irregularly Spaced Data

```
interp(x, y, z, xo, yo, ncp, extrapol)
```

ARGUMENTS:

x: x-coordinates of data points.
y: y-coordinates of data points.
z: z-coordinates of data points.
xo: vector of x-coordinates of output grid. Default, 40 points evenly spaced over the range of *x*.
yo: vector of y-coordinates of output grid. Default, 40 points evenly spaced over the range of *y*.
ncp: number of additional points to be used in computing partial derivatives at each data point. If *ncp* is zero, linear interpolation will be used in the triangles bounded by data points. Otherwise, *ncp* must be 2 or greater, but smaller than the number of data points. (Cubic interpolation is done if partial derivatives used. Default 0.)
extrapol: logical flag, should extrapolation be used outside of the convex hull determined by the data points? Default FALSE. No extrapolation can be performed if *ncp* is zero.

VALUE:

structure with 3 components:

x: vector of x-coordinates of output grid, the same as input argument *xo*.

y: vector of y-coordinates of output grid, the same as input argument *yo*.

z: matrix of fitted z-values. The value $z[i,j]$ is computed at the x,y point $x[i],y[j]$.

If *extrap* is FALSE, z-values for points outside the convex hull are returned as NA. The resulting structure is suitable for input to the function *contour*.

REFERENCE:

Hiroshi Akima, "A Method of Bivariate Interpolation and Smooth Surface Fitting for Irregularly Distributed Data Points", *ACM Transactions on Mathematical Software*, Vol 4, No 2, June 1978, pp 148-164.

EXAMPLES:

```
fit ← interp(x,y,z)    # fit to irregularly spaced data
contour(fit)           # contour plot
```

knapsack: (see **napsack**)

l1fit: Minimum Absolute Residual (L1) regression

`l1fit(x, y, int)`

ARGUMENTS:

x: X matrix for fitting $Y = Xb + e$ with variables in columns, observations across rows. Should not contain column of 1's (see argument *int*). Number of rows of x should equal the number of data values in y. There should be fewer columns than rows.

y: numeric vector with as many observations as the number of rows of x.

int: flag for intercept; if TRUE (default) an intercept term is included in regression model.

VALUE:

structure defining the regression (compare function *regress*).

coef: vector of coefficients with constant term as first value (optional depending on argument *int*)

resid: residuals from the fit, i.e. *resid* is $Y - Xb$.

REFERENCE:

Barrodale and Roberts, *CACM* (June 1974) pp 319-320 "Solution of an Overdetermined System of Equations in the L1 Norm".

labclust: Label a Cluster Plot

`labclust(x, y, label)`

ARGUMENTS:

x,y: coordinates of the leaves of the tree, as produced by *plclust*.

label: optional control of labels on the cluster leaves. By default, leaves are labelled with object number. If *label* is a character vector, it is used to label the leaves. Otherwise it is interpreted as a logical flag (in particular, the value FALSE suppresses any labelling of leaves, as is appropriate with large trees).

Graphical parameters may also be supplied as arguments to this function (see *par*).

EXAMPLES:

```
xy←plclust(tree,plot=FALSE)    # save coords of tree
plclust(tree,label=FALSE)    # plot it
lablcust(xy,label=names)    # now label it
```

lag: Create a Lagged Time Series

```
lag(x, k)
```

ARGUMENTS:

x: a time series. Missing values (NAs) are allowed.
k: the number of positions the new series is to lag the input series (default=1); negative value will lead the series.

VALUE:

a time series of the same length as *x* but lagged by *k* positions. Only the start and end dates are changed; the series still has the same number of observations.

See also *tsmatrix* for aligning the time domains of several series.

EXAMPLES:

```
l12gnp←lag(gnp,12) # gnp lagged by 12 months
```

leaps: All-subset Regressions by Leaps and Bounds

```
leaps(x, y, wt, int, method, nbest, names)
```

ARGUMENTS:

x: matrix of independent variables. Each column of *x* is a variable, each row an observation. There should be fewer columns than rows.
y: vector of dependent variable with the same number of observations as the number of rows of *x*.
wt: optional vector of weights for the observations.
int: logical flag, should an intercept term be used in the regressions? Default TRUE.
method: character string describing the method used to evaluate a subset. Possible values are "Cp", "r2", and "adjr2" corresponding to Mallows Cp statistic, r-square, and adjusted r-square. Only the first character need be supplied. Default, "Cp".
nbest: integer describing the number of "best" subsets to be found for each subset size. In the case of r2 or Cp methods, the *nbest* subsets (of any size) are guaranteed to be included in the output (but note that more subsets will also be included). Default 10.
names: optional character vector giving names for the independent variables. Default, the names are 1, 2, ... 9, A, B, ...

VALUE:

structure with four components.

Cp: the first returned component will be named "Cp", "adjr2", or "r2" depending on the method used for evaluating the subsets. This component gives the values of the desired statistic.
size: the number of independent variables (including the constant term if *int* is TRUE) in each subset.
label: a character vector, each element giving the names of the variables in the subset.

which: logical matrix with as many rows as there are returned subsets. Each row is a logical vector that can be used to select the columns of *x* in the subset.

REFERENCE:

George M. Furnival and Robert W. Wilson, Jr., "Regressions by Leaps and Bounds", *Technometrics*, Vol 16, No 4, November 1974, pp 499-511.

EXAMPLES:

```
r←leaps(x,y)
plot(r$size,r$Cp,type="n")
text(r$size,r$Cp,r$label)    # produces Cp plot
regress(x[,r$which[3,]],y)   # regression corresponding
                             # to third subset
```

len: Length of a vector

```
len(x)
```

ARGUMENTS:

x: vector structure. Missing values (NAs) are allowed.

VALUE:

the number of data values in *x*.

lgamma: (see **gamma**)

lines points: Add Lines or Points to Current Plot

```
lines(x, y)
points(x, y, mark=)
```

ARGUMENTS:

x,y: the co-ordinates for lines or points. A time series or structure containing *x* and *y* may also be given for *x*. Missing values (NAs) are allowed.

mark=: mark number for plotting special symbols at the points. Basic marks are: square (0); octagon (1); triangle(2); cross(3); X (4); diamond(5) and inverted triangle(6). To get superimposed versions of the above use the following arithmetic(!): 7==0+4; 8==3+4; 9==3+5; 10==1+3; 11==2+6; 12==0+3; 13==1+4; 14==0+2.

Graphical parameters may also be supplied as arguments to this function (see *par*).

Data values with an NA in either *x* or *y* are not plotted by *points*. Also, *lines* does not draw to any such point, thus giving a break in the line segments.

If a log scale was specified for either the *x*- or *y*-axis, *points* and *lines* will plot the corresponding values on a log scale.

EXAMPLES:

```
par(usr=c(-1,15,0,1)) # produce a plot of all the marks
for(i in 0:14){
```



```

points(i,.5,mark=i)
text(i,.35,i)}
text(7,.75,'Samples of "mark=" Parameter')

```

Sample of "mark=" Parameter

□	○	△	+	×	◇	▽	⊠	*	◆	⊕	⊗	⊞	⊠	⊠
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

list ls: List of Datasets in Database

```

list(pattern, pos)
ls(pattern, pos)

```

ARGUMENTS:

pattern: an optional character string describing the dataset names of interest. For example, *list("abc")* or *ls("abc")* would only return the name of the dataset *abc* (if it existed). The character "*" at the end of the pattern matches any number of characters following. *list("abc*")* would return the names of any datasets whose names began with *abc*. The default pattern is "*", which matches all dataset names.

pos: the position on the database search list of the database to be searched. Position 1 (default) is the working database. Normally, positions 2 and 3 are the save and shared data bases.

VALUE:

a character vector which is the alphabetical list of the datasets (matching pattern) on the specified database.

EXAMPLES:

```

list      #list all working dataset names
list("lottery*",pos=3) #shared db names beginning lottery

```

If a prefix is in effect, all matching takes place within datasets whose name begins with the prefix. This can be changed by using an initial "\$" in the pattern, thus matching "fully qualified" dataset names. The pattern "\$*" matches all dataset names, regardless of current prefix, while "*" matches all names within the prefix.

plnorm qlnorm rlnorm: Log-normal Probability Distribution

```
plnorm(q, par1, par2)
qlnorm(p, par1, par2)
rlnorm(n, par1, par2)
```

ARGUMENTS:

q: vector of quantiles.
p: vector of probabilities.
par1,par2: vectors of means and standard deviations of the distribution of the log of the random variable. Thus, *exp(par1)* is a scale parameter and *par2* a shape parameter for the lognormal distribution. Default *par1=0,par2=1*.
n: sample size.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*rlnorm*).

log: (see *exp*)

log10: (see *exp*)

< <= > >= == != : Logical Operators

```
expr1 op expr2
```

ARGUMENTS:

expri: numeric structure.

VALUE:

vector structure with FALSE or TRUE in each element according to the truth of the element-wise compare of the operands. For vector operands, the value is as long as the longer of *expr1* and *expr2*. For time-series operands, the time domain of the result is the intersection of the domains of the operands.

See also *xor* for exclusive-or function.

EXAMPLES:

```
a>b      # true when a greater than b
x[x>100]  # all x with values larger than 100
state=="Wyoming" # TRUE or FALSE
```


& (and) | (or) ! (not): Logical Operations

```

expr1 & expr2
expr1 | expr2
! expr1

```

ARGUMENTS:

expri: logical structures. Missing values (NAs) are allowed.

VALUE:

logical result of and-ing, or-ing or negation. In case of the first two the result is as long as the longer of the operands. When time series operands are used, the time domain of the value is the intersection of the time domains of the operands.

EXAMPLES:

```
x[a>13 & b<2]    #elements of x corresp. to a>13 and b<2
```

plogis qlogis rlogis: Logistic Probability Distribution

```

plogis(q, par1, par2)
qlogis(p, par1, par2)
rlogis(n, par1, par2)

```

ARGUMENTS:

q: vector of quantiles.
p: vector of probabilities.
par1: vector of location parameters. Default is 0.
par2: vector of scale parameters. Default is 1.
n: sample size.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*rlogis*).

loglin: Contingency Table Analysis

```
loglin(table, margin, start, eps, iter, print)
```

ARGUMENTS:

table: contingency table (array) to be fit by log-linear model. Table values must be non-negative.
margins: vector describing the marginal totals to be fit. A margin is described by the factors not summed over, and margins are separated by zeroes. Thus c(1,2,0,3,4) would indicate fitting the 1,2 margin (summing over variables 3 and 4) and the 3,4 margin in a four-way table.
start: starting estimate for fitted table. If *start* is omitted, a start is used that will assure convergence. If structural zeroes appear in *table*, *start* should contain zeroes in corresponding entries, ones in other places. This assures that the fit will contain those zeroes.
eps: maximum permissible deviation between observed and fitted marginal totals. Default 0.1.
iter: maximum number of iterations, default 20.
print: flag; if TRUE (the default), the final deviation and number of iterations will be printed.

VALUE:

structure like *table*, but containing fitted values.

REFERENCE:

S. J. Haberman, "Log-linear Fit for Contingency Tables - Algorithm AS51", *Applied Statistics* (1972), Vol 21, No 2, pp 218-225.

logo: Draw the S Logo

logo(x, y, size)

ARGUMENTS:

x,y: optional co-ordinates for the center of the logo. By default, plotted in lower right of figure.
size: optional size of logo relative to the height of a character. Default 2.5.

Graphical parameters may also be supplied as arguments to this function (see *par*).

EXAMPLES:

logo # logo in lower right hand corner of figure

lowess: Scatter Plot Smoothing

lowess(x, y, f, iter, delta)

ARGUMENTS:

x,y: vectors of data for scatter plot.
f: fraction of data used for smoothing at each x point. The larger the *f* value, the smoother the fit. Default 2/3.
iter: number of iterations used in computing robust estimates. Default 3.
delta: interval size (in units corresponding to x). If *lowess* estimates at two x values within *delta* of one another, it fits any points between them by linear interpolation. Default 0, implying all but identical x values are estimated independently.

VALUE:

plot structure containing components named *x* and *y* which are the x,y points of the smoothed scatter plot. Note that *x* is a sorted version of the input *x* vector, with duplicate points removed.

This function may be slow for large numbers of points; execution time is proportional to $(\text{iter} * f * n^2)$. Increasing *delta* may speed things up, as will decreasing *f*.

REFERENCE:

W. S. Cleveland, "Robust Locally Weighted Regression and Smoothing Scatterplots", *JASA*, Vol 74, No 368, pp 829-836., December 1979.

EXAMPLES:

plot(x,y);lines(lowess(x,y)) #scatter plot with smooth
fit ← lowess(x,y); resid ← y-approx(fit,x) #residual from smooth

ls: (see list)

macros

Macros can be used to reduce the amount of typing to perform repetitive tasks within S. A macro is simply a set of commands which will be executed as if they were typed from the terminal. The power of macros lies in the ability to give them arguments, so that similar tasks may be carried out by a single macro.

A macro is often used as a repository for a set of commands that is long or complicated, hence hard to type interactively. Because it involves a certain amount of overhead, the macro processor is ordinarily not used when input is typed to S. The special character "?" before a name signals a macro and causes the current line to be sent through the macro processor before being executed.

Macros are defined by means of the *define* function. The *medit* function can be used to edit macros and the *mprint* function to print them.

mail: Suggestions, Questions and Reports

mail
mail(file)

ARGUMENTS:

file: optional character string giving the name of a file containing a letter. Without an argument, *mail* will prompt (with the characters "M>") for a multi-line letter to be sent to the local S support staff. The letter is terminated by a line consisting only of a period.

MAKE: (see CHAPTER)

match: Match Items in Vector

match(x, list)

ARGUMENTS:

x: vector of items that are to be looked for in *list*.
list: the possible values in *x*.

VALUE:

vector like *x* giving the index in *list* of the item that matches each element of *x*. If an element of *x* is not in *list*, NA is returned.

EXAMPLES:

```
match(data,primes)
state.abb[match(names,state.name)] #change names to abbrevs
```

matlines: (see **matplot**)

%*: Matrix Multiplication Operator

```
mat1 %* mat2
```

ARGUMENTS:

mat1: matrix or vector.

VALUE:

matrix product of *mat1* and *mat2*.

Vector results are returned as vectors, not as (n by 1) or (1 by n) matrices.

The last extent of *mat1* must be the same size as the first extent of *mat2*. Vectors are not oriented, therefore a vector of length n can multiply an n by n matrix on the left or right.

matplot matlines matpoints: Plot Columns of Matrices

```
matplot(x, y, type=, lty=, pch=, col=)
```

ARGUMENTS:

x,y: vectors or matrices of data for plotting. The first column of *x* is plotted against the first column of *y*, the second column of *x* against the second column of *y*, etc. If one matrix has fewer columns, plotting will cycle back through the columns again. (In particular, either *x* or *y* may be a vector, against which all columns of the other argument will be plotted.)

type=: an optional character string, telling which type of plot (points, lines, both, none or high-density) should be done for each plot. The first character of *type* defines the first plot, the second character the second, etc. Elements of *type* are cycled through; e.g., "pl" alternately plots points and lines. Default is "p" (points) for *matplot* and *matpoints* and "l" for *matlines*.

lty=: optional vector of line types. The first element is the hardware line type for the first line, etc. Line types will be used cyclically until all plots are drawn. Default is 1,2,3,4,5.

pch=: optional character vector for plotting-characters (only the first element is used). The first character is the plotting-character for the first plot, the second for the second, etc. Default is the digits (1 through 9, 0) then the letters.

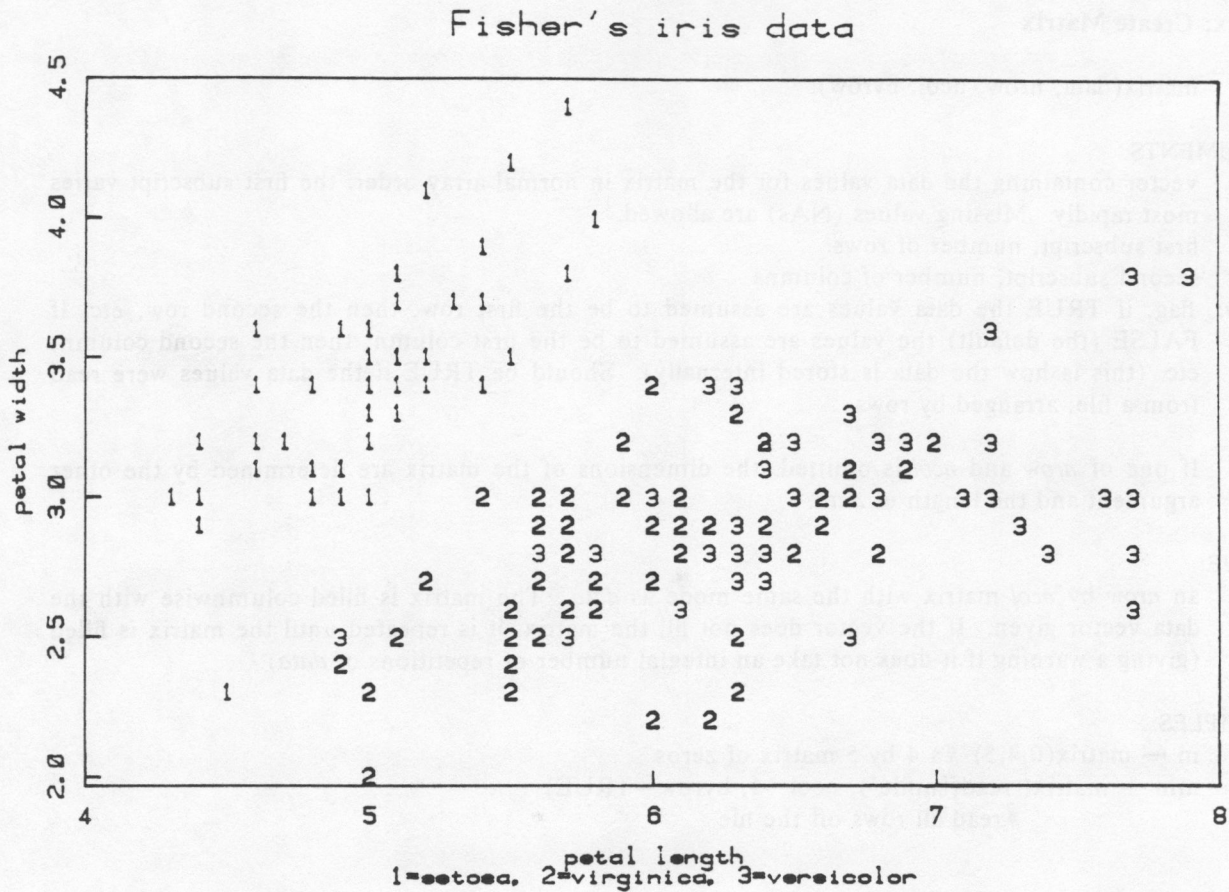
col=: optional vector of colors. Default is 1,2,3,4. Colors are also cycled if necessary.

Graphical parameters may also be supplied as arguments to this function (see *par*).

matplot generates a new plot; *matpoints* and *matlines* add to the current plot.

EXAMPLES:

```
matplot(x,y,type="pl") # points for 1st col, lines for 2nd
matpoints(x,y,pch="*") # points with "*" for all plots
# the example plot is produced by:
matplot(iris[,1,],iris[,2,],xlab="petal length",
ylab="petal width",
sub="1=setosa, 2=virginica, 3=versicolor",
main="Fisher's iris data")
```

matpoints: (see matplot)

%c: Matrix Cross Product Operator

`mat1 %c mat2`

ARGUMENTS:

mat1: matrix or vector.

VALUE:

matrix representing the cross product of *mat1* and *mat2*, defined as $t(mat1) \%* mat2$, where $\%$ is matrix multiplication and t is transposition.

matrix: Create Matrix

```
matrix(data, nrow, ncol, byrow)
```

ARGUMENTS:

data: vector containing the data values for the matrix in normal array order: the first subscript varies most rapidly. Missing values (NAs) are allowed.

nrow: first subscript, number of rows.

ncol: second subscript, number of columns

byrow: flag, if TRUE the data values are assumed to be the first row, then the second row, etc. If FALSE (the default) the values are assumed to be the first column, then the second column, etc. (this is how the data is stored internally). Should be TRUE if the data values were read from a file, arranged by rows.

If one of *nrow* and *ncol* is omitted, the dimensions of the matrix are determined by the other argument and the length of *data*.

VALUE:

an *nrow* by *ncol* matrix with the same mode as *data*. The matrix is filled columnwise with the data vector given. If the vector does not fill the matrix, it is repeated until the matrix is filled (giving a warning if it does not take an integral number of repetitions of *data*).

EXAMPLES:

```
m ← matrix(0,4,5) #a 4 by 5 matrix of zeros
mm ← matrix( read("mfile"), ncol=5, byrow=TRUE)
      #read all rows off the file
```

max min: Extremes

```
max(arg1, arg2, ...)
min(arg1, arg2, ...)
```

ARGUMENTS:

argi: numeric structure. Missing values (NAs) are allowed.

VALUE:

the single maximum or minimum value found in any of the args.

See also *pmax*, *pmin* for generating vector of parallel extremes.

mdscale: Multi-dimensional Scaling

```
mdscale(d, k, eig)
```

ARGUMENTS:

d: distance matrix structure. Normally this will be the result of the function *dist*, but it can be any data of the form returned by *dist*, or a full (symmetric) matrix.

k: dimensionality of the output space. Default 2.

eig: logical flag which controls return of eigenvalues used in algorithm. These eigenvalues can give information about appropriate dimensionality of solution. Default, FALSE.

VALUE:

if argument *eig* is TRUE, a structure with two components named *eig* and *points*. Otherwise, a matrix with *k* columns and as many rows as there were objects whose distances were given in *d*. Row *i* gives the coordinates in *k*-space of the *i*-th object.

eig: vector of eigenvalues (as many as original data points).

points: matrix of fitted data points described above.

REFERENCE:

Metric multi-dimensional scaling described by Warren S. Torgerson, *Theory and Methods of Scaling*, pp. 254-259, Wiley (1958).

EXAMPLES:

```
x←mdscale(dist) # default 2-space
coord1←x[,1]; coord2←x[,2]
par(pty="s") # set up square plot
r←range(coord1,coord2) # get overall max, min
plot(coord1,coord2,type="n",xlim=r,ylim=r) # set up plot
# note units per inch same on x and y axes
text(coord1,coord2,seq(coord1)) # plot integers
```

mean: Mean Value

```
mean(x, trim)
```

ARGUMENTS:

x: numeric structure.

trim: fraction of values to be trimmed off each end of the ordered data. Default value 0.

VALUE:

(Trimmed) mean of *x*.

EXAMPLES:

```
mean(scores,trim=.25) # trims outer 50% of data
```

median: Median

```
median(x)
```

ARGUMENTS:

x: numeric structure. Missing values (NAs) are allowed.

VALUE:

median of data (excluding the NAs).

medit: Edit Macros

medit(defin, pos)

ARGUMENTS:

defin: the macro to be edited. Note that macros are datasets with names beginning "mac."
pos: optional, database position on which edited macro is to be saved; default 2.

medit invokes the *ed* text editor on a copy of the macro text. Use *ed* to make whatever changes are desired in the macro. Then use the *w* command in *ed* to write the macro, followed by *q* to exit from *ed*. The macro will automatically be re-processed through the *define* function, and stored on the specified database.

EXAMPLES:

medit(mac.abc) # edits macro *abc* and replaces it

min: (see **max**)

mode: Mode (Data Type) of the Values in a Vector

mode(x)

ARGUMENTS:

x: vector structure. Missing values (NAs) are allowed.

VALUE:

the mode of *x*. Values are LGL=1, INT=2, REAL=3, CHAR=5.

NOTE:

this function has nothing to do with the statistical concept of the mode of a distribution.

mprint: Print Macros

mprint(mac1, mac2, ..., list=, file=, pos=)

ARGUMENTS:

mac1: macro to be printed. Note that macros all have names beginning "mac."
list=: character vector giving the names of macros to be printed.
file=: character string name of file to which the macro printing is directed. Default is the user's terminal.
pos=: position on database search list where datasets given in *list* will be found. Default 0, search for each dataset on all attached databases.

The arguments *mac1*, ... and *list=* are mutually exclusive.

The printed macros have macro substitution characters *\$1*, etc. in the body of the macro instead of the named arguments.

EXAMPLES:

mprint(mac.abc) # print macro *abc* on terminal


```
mprint(list=list("mac.*",pos=2),file="abcd",pos=2)
# print all macros on database onto file abcd
```

mprompt: Produce Shell of Documentation for Macros

```
mprompt(macro, file)
```

ARGUMENTS:

macro: dataset containing the macro to be documented, i.e., *mac.abc*.

file: optional name of output documentation file, default is *abc.d*, where *abc* is the name of the macro.

This function is ordinarily used via the system macro "?prompt".

EXAMPLES:

```
mprompt(mac.def) # document macro def, produces file def.d
```

mstr: Modify Structure

```
mstr(str, arg2, arg3, ...)
```

ARGUMENTS:

str: arbitrary structure to be modified.

argi: arbitrary name=value argument.

VALUE:

the structure *str* modified by the named arguments. If a name matches a component name of *str*, that entry in *str* is changed to the value of the argument. An argument with a null value will delete the entry. If a name does not match any entries in *str*, the entry will be added to *str* in the result. Missing values (NAs) are allowed.

mstree: Multivariate Planing and Lining from Minimal Spanning Tree

```
mstree(x, plane)
```

ARGUMENTS:

x: matrix of data where rows correspond to observations, columns to variables. Should be scaled so that values on all variables are roughly comparable (the algorithm computes Euclidean distances from one observation to another).

plane: logical, should multivariate planing and lining information be returned? If TRUE (the default), all components listed below are returned; if FALSE, only the minimum spanning tree *mst* is returned.

VALUE:

structure with components *x*, *y*, *mst*, and *order*, describing the planing, minimal spanning tree, and two versions of lining. If *plane* is FALSE, only the *mst* vector is returned.

x,y: coordinates of the observations computed by the Friedman- Rafsky algorithm.

mst: vector of length *nrow(x)-1* describing the edges in the minimal spanning tree. The *i*th value in this vector is an observation number, indicating that this observation and the *i*th observation should be linked in the minimal spanning tree.

order: matrix, $nrow(x)$ by 2, giving two types of ordering: The first column presents the standard ordering from one extreme of the MST to the other. The second column presents the radial ordering, based on distance from the center of the MST.

REFERENCE:

Friedman, J. H. and Rafsky, L. C. "Graphics for the multivariate two-sample problem", *Stanford Linear Accelerator Corp.* SLAC PUB-2193 (1978).

EXAMPLES:

```
tree ← mstree(data)
plot(tree)
mst ← tree$mst; s ← seq(mst)
x ← tree$x; y ← tree$y
segments(x[s],y[s],x[mst],y[mst])
```

mtext: Text in the Margins of a Plot

```
mtext(text, side, line, outer, at)
```

ARGUMENTS:

text: character string to be plotted.
side: side (1,2,3,4 for bottom, left, top, or right).
line: line (measured out from the plot in units of standard-sized character heights). By default (unless parameter *mar* is changed by the *par* function), standard-sized characters can be placed at values of *line* less than or equal to 4 on the bottom, 3 on left and top, and 1 on the right.
outer: logical flag, should plotting be done in outer margin? Default FALSE.
at: optional vector of positions at which *text* is to be plotted. If *side* is 1 or 3, *at* will represent x-coordinates. If *side* is 2 or 4, *at* will represent y-coordinates. *at* can be used for constructing specialized axis labels, etc.

mulbar: Multiple Bar Plot

```
mulbar(width, height, rowlab, collab, gap)
```

ARGUMENTS:

width: matrix of bar widths, all values must be positive.
height: matrix of bar heights, values may be negative. *width* and *height* must have the same number of rows and columns.
rowlab: optional character vector for row labels.
collab: optional character vector for column labels.
gap: fraction of plot used for gap between rows and columns, default .1

Graphical parameters may also be supplied as arguments to this function (see *par*).

EXAMPLES:

```
mulbar(sqrt(resid),resid/sqrt(fit))
# chi plot for contingency table
```


ncol nrow: Extents of a Matrix

```
ncol(x); nrow(x)
```

ARGUMENTS:

x: matrix. Missing values (NAs) are allowed.

VALUE:

the number of rows or columns in *x*.

ncomp: Number of Components

```
ncomp(x)
```

ARGUMENTS:

x: hierarchical structure

VALUE:

the number of components in *x*.

Time-series and arrays have 2 components.

EXAMPLES:

```
for(i in seq(ncomp(x))) tsplot(x$[i])
#time-series plot of each component of structure x
```

NEWDOC: (see **PROMPT**)**pnorm qnorm rnorm: Normal Probability Distribution**

```
pnorm(q, par1, par2)
qnorm(p, par1, par2)
rnorm(n, par1, par2)
```

ARGUMENTS:

q: vector of quantiles.
p: vector of probabilities.
par1: vector of means. Default is 0.
par2: vector of standard deviations. Default is 1.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*rnorm*).

EXAMPLES:

```
rnorm(20,0,10) #sample of 20, mean 0, standard dev. 10
```

na NA: Pick Out Missing Values

na(x)
NA(x)

ARGUMENTS:

x: vector structure.

VALUE:

logical structure like *x*, with TRUE wherever a missing value (NA) appeared in *x*, and FALSE elsewhere.

NOTE:

Whenever an NA appears in either operand of any logical operation, the corresponding value is NA. In particular, $x == NA$ will always produce a vector of all NAs; the *na* function must be used to test for missing values.

NA: (see *na*)

napsack: Solve Knapsack Problems

napsack(x, target, best)

ARGUMENTS:

x: vector of generators for knapsack problem. The function attempts to find subsets of *x* whose sums are equal (or close to) *target*.

target: scalar target value.

best: the desired number of solutions (or approximate solutions) to the problem. Default 10.

VALUE:

matrix of logical values, of size *len(x)* by *best*. Each column tells which elements of *x* are contained in one of the *best* subsets. Thus, a column of the result can be used to subscript *x* to obtain a subset.

The knapsack problem is NP-complete, and the algorithm is exponential in time and space complexity. If *n* is the length of *x*, then the algorithm requires time $O(2^{n/2})$ and space $O(2^{n/4})$. Problems with $n < 30$ can be readily solved by this function. Remember that both time and space requirements increase very rapidly with problem size!

The solutions produced may not include all subsets that can generate solutions with a certain error. It is guaranteed to produce an exact solution if one is possible, but may not find all of a number of exact solutions.

REFERENCE:

Richard Schroepel and Adi Shamir, "A $T=O(2^{n/2})$, $S=O(2^{n/4})$ Algorithm for Certain NP-Complete Problems".

EXAMPLES:

```
# given areas of counties of Nevada, find subsets of the
# counties with approximately 1/2 the total state area
subsets ← napsack(nevada,sum(nevada)/2)
subsets %c nevada # areas of the subsets
```


ncol nrow: Extents of a Matrix

`ncol(x); nrow(x)`

ARGUMENTS:

x: matrix. Missing values (NAs) are allowed.

VALUE:

the number of rows or columns in *x*.

ncomp: Number of Components

`ncomp(x)`

ARGUMENTS:

x: hierarchical structure

VALUE:

the number of components in *x*.

Time-series and arrays have 2 components.

EXAMPLES:

```
for(i in seq(ncomp(x))) tsplot(x[i])
# time-series plot of each component of structure x
```

NEWDOC: (see **PROMPT**)**pnorm qnorm rnorm: Normal Probability Distribution**

```
pnorm(q, par1, par2)
qnorm(p, par1, par2)
rnorm(n, par1, par2)
```

ARGUMENTS:

q: vector of quantiles.
p: vector of probabilities.
par1: vector of means. Default is 0.
par2: vector of standard deviations. Default is 1.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*rnorm*).

EXAMPLES:

```
rnorm(20,0,10) # sample of 20, mean 0, standard dev. 10
```

nper: (see end)

nproc: Reset Limit for Number of Simultaneous Processes

nproc(n)

ARGUMENTS:

n: number of S functions allowed to be simultaneously active processes. Increasing this number, when the system is lightly loaded, can lead to more rapid response, if the user is re-calling the same S functions repeatedly. However, there is a danger of degrading overall performance if *n* is too large. There will likely be a local limit to *n*; its value is a delicate and (currently) poorly understood issue to be considered by local UNIX and S gurus. By default, *n* is 5.

nrow: (see ncol)

%x: Operators (special)

expression %x expression

Special operators are available within S to allow functions of two arguments to be written in infix notation. These special operators all begin with a percent (%) sign, and are followed by any character. All special operators are defined to have very high precedence, so that they tend to be performed only upon adjacent names, for example *-a %* b+3* would carry out matrix multiplication of *a* and *b*, negate the result, and then add 3 (see *precedence*).

The only requirement for adding new special operators is that the function take two arguments, and that the command name be of the %x form.

Examples of special operators are %% (mod), %* (matrix-multiply), %o (outer product), %m (coerce) and %c (cross-product).

option: (see options)

options option: Set Options

options(echo=, width=, length=, space=, max=, macsize=, dump=, sourcexit=)

ARGUMENTS:

echo=: controls the amount of information echoed by S. If *echo=0* (the default) there is no printing of commands or macro expansions. If *echo=1*, there is printing of all macro expansions and input from source files. If *echo=2* (the default for batch mode) all commands are printed before being executed. If *echo=3*, commands, input from source files, and macro expansions are printed.

width=: specifies the width (in print positions) of the user's terminal. Default 80, minimum 20, maximum 133.

length=: the length of an output page, currently used only for the length of printer plots. Default 56, must be between 20 and 100.

space=: the number of characters separating printed numbers. Default 2.

max=: the maximum number of elements of any vector that will be printed. If the length of a vector to be printed exceeds this number, only the first *max* will be printed, followed by a message telling what the actual length of the vector was. Default is 1000000.

macsize=: the size (in characters) of the macro workspace. Should be large enough to accommodate the largest macro used, and larger if macros are nested. Default 2000.

dump=: how long should an expression be (in characters) before an error or interrupt generates an automatic dump (see *edit*). Default 20.

sourcexit=: if TRUE, exit from a source file if an error occurs (the default); if FALSE, continue execution of the source file regardless of errors.

If *options* is called without arguments, it prints the current options values.

order: Ordering to Create Sorted Data

`order(x1, x2, ...)`

ARGUMENTS:

xi: vector. All arguments must have the same length.

VALUE:

integer vector with same number of elements as data elements in *xi*. Contains the indices of the data elements in ascending order, i.e., the first integer is the subscript of the smallest data element, etc. For character vectors, the sorting order is lexicographic.

Sorting is primarily based on vector *x1*. Values of *x2* are used to break ties on *x1*, etc. All sorting is done in ascending order.

EXAMPLES:

```
list ← order(x); cbind(x[list],y[list],z[list])
#sort y and z to follow the order of x
```

This function is often used in conjunction with subscripting for sorting several parallel arrays.

outer %o: Generalized Outer Products

`x %o y` #operator form
`outer(x, y, fun, arg1, arg2, ...)` #general form

ARGUMENTS:

x,y: first and second arguments to the function *fun*.

fun: in the general form, the name of some S function. This function will be called repeatedly, for all the values of the first subscript of *x* and the last subscript of *y*. For example, if *x* and *y* are vectors, and *fun* returns a single value when its two arguments are single values, the result is a matrix of *len(x)* by *len(y)*. In the operator case, *fun* defaults to multiply (*).

argi: other arguments to *fun*, if needed.

VALUE:

array, whose dimension vector is the concatenation of the dimension vectors of *x* and *y*.

EXAMPLES:

```

z ← x %o y # z[i,j] == x[i] * y[j]
z ← outer(x,y,"/") # z[i,j] == x[i]/y[j]

```

pairs: Plot Pair-wise Scatters of Multivariate Data

pairs(x, labels, type, head, full, max, text)

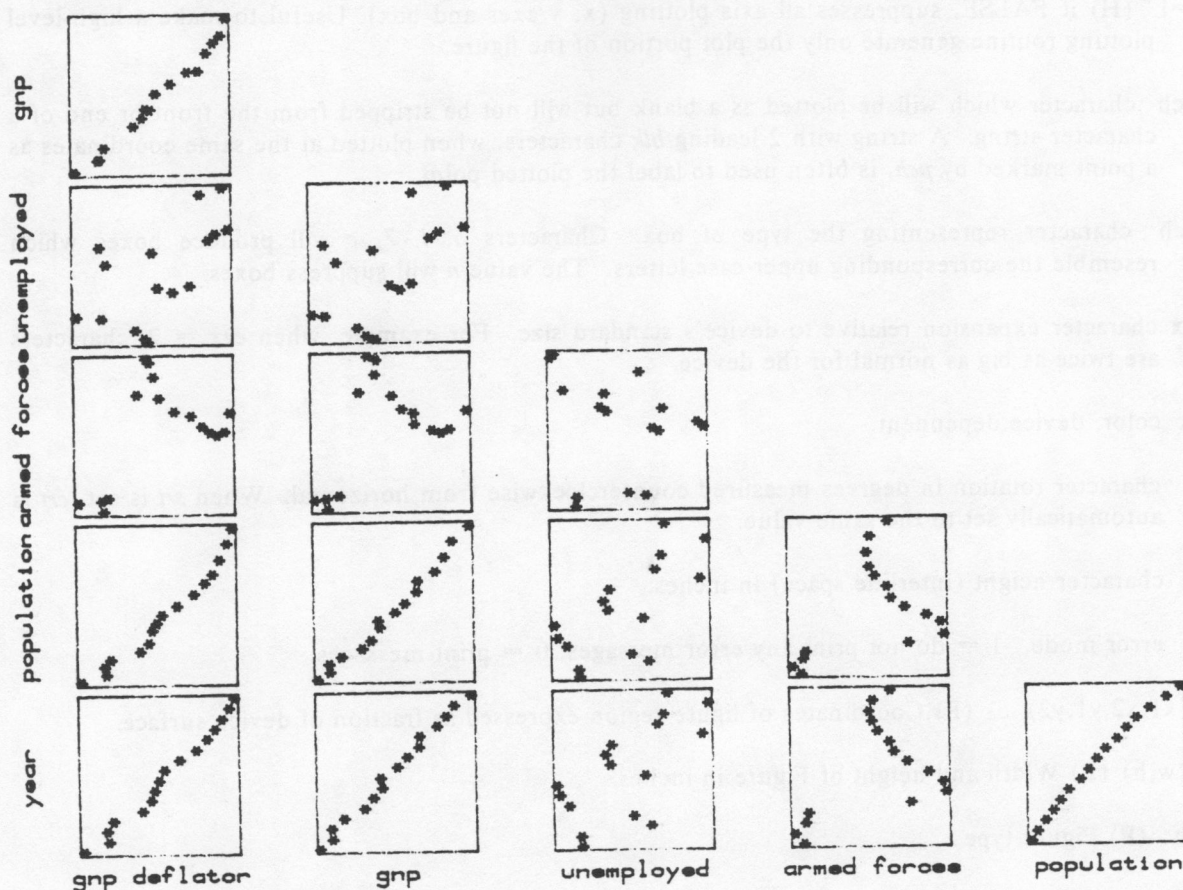
ARGUMENTS:

- x:** matrix of data to be plotted. Missing values (NAs) are allowed. A scatter plot will be produced for each pair of columns of *x*.
- labels:** optional character vector for labelling the *x* and *y* axes of the plots. The strings *labels[1]*, *labels[2]*, etc. are the labels for the 1st, 2nd, etc. columns of *x*. By default, labels are "1", "2", etc. If supplied, the label vector must have length equal to *ncol(x)*.
- type:** optional character string to define the type of the scatter plot. Possible values are "p", "l", "b" for points, lines or both. Default is "p". See also *text* below.
- head:** optional character string for a running head. This is plotted as a title at the top of each page. By default, the name of the data is used. If there is more than one page, the page number is included in the title.
- full:** if TRUE, the full square array of plots is produced. Otherwise, only the lower triangle (saving space, but making interpretation harder). Default FALSE.
- full:** should the full *ncol(x)-1* by *ncol(x)-1* array of plots be produced? Default FALSE (only the lower triangle).
- max:** optional, suggested limit for the number of rows or columns of plots on a single page. Default forces all plots on 1 page. The algorithm tries to choose an array of plots which efficiently uses the available space on the display.
- text:** optional vector of text to be plotted at each of the points. If the length of the vector is less than *nrow(x)*, elements will be reused cyclically. If missing, the plotting is controlled by *type*.

EXAMPLES:

```
pairs(longley.x, labels=longley.collab, head="Longley Data")
```


Longley Data

**par: Graphical Parameters**

```
par(arg1, arg2, ...)
```

ARGUMENTS:

argi: arguments in the name=value form, where name is one of the options below.

These are the graphical parameters which control the appearance of graphical output. There are three classes of parameters: those which can appear in *par* or any graphical function; those which are available only through the *par* command; and those which are available only through the high-level operations (*plot*, *qqplot*, etc). If given through *par*, the options are set permanently; otherwise, they are reset at the end of the graphical operation.

Commonly used parameters are: *pch*=plotting character, *cex*=character expansion, *lty*=line type, *type*=type of plot, *log*=logarithmic axes, *[xy]lim*=range for x or y axis, *main*=main title, *[xy]lab*=x or y axis label.

PARAMETERS:

In the following list of parameters, those marked (P) are available only from calls to *par*. Those marked (H) are available only through high-level graphical functions. The argument *ch* denotes a character, *s* is a character string, *i*, *j*, *m*, and *n* are integers, *l* is a logical value, and the rest are numeric.

adj=x string justification parameter. 0 = left justify, 1 = right justify, .5 = center.

`axes=1` (H) if FALSE, suppresses all axis plotting (x, y axes and box). Useful to make a high-level plotting routine generate only the plot portion of the figure.

`blk=ch` character which will be plotted as a blank but will not be stripped from the front or end of a character string. A string with 2 leading *blk* characters, when plotted at the same coordinates as a point marked by *pch*, is often used to label the plotted point.

`bty=ch` character representing the type of box. Characters *o*, *l*, *7*, *c* will produce boxes which resemble the corresponding upper-case letters. The value *n* will suppress boxes.

`cex=x` character expansion relative to device's standard size. For example, when *cex* = 2, characters are twice as big as normal for the device.

`col=x` color, device dependent.

`crt=x` character rotation in degrees measured counterclockwise from horizontal. When *srt* is set, *crt* is automatically set to the same value.

`csi=x` character height (interline space) in inches.

`err=x` error mode, -1 = do not print any error messages, 0 = print messages.

`fig=c(x1,x2,y1,y2)` (P) Coordinates of figure region expressed as fraction of device surface.

`fin=c(w,h)` (P) Width and height of Figure in inches.

`fty=ch` (P) Figure type.

`lab=c(x,y)` desired number of tick intervals on the X and Y axes. and the length of labels on both axes.

`las=x` style of axis labels. 0 = always parallel to axis, 1 = always horizontal, 2 = always perpendicular to axis.

`log=ch` (H) Controls logarithmic axes. Values 'xy', 'x', or 'y' produce log-log or log-x or log-y axes.

`lty=x` line type, device dependent. Normally type 1 is solid, 2 and up are dotted or dashed.

`mai=c(x1,x2,x3,x4)` (P) Margin size specified in inches. Values given for bottom, left, top, and right margins in that order.

`main=s` (H) Main title for top of plot. Plotted in characters of size $1.5 * cex$.

`mar=c(xbot,xlef,xtop,xrig)` (P) Maximum value for margin coordinates on each side of plot. Margin coordinates range from 0 at the edge of the box outward in units of *mex* sized characters. If the margin is respecified by *mai* or *mar*, the plot region is re-created to provide the appropriate sized margins within the figure. Default values are (5.1,4.1,4.1,2.1).

`mex=x` (P) The coordinate unit for addressing locations in the margin is expressed in terms of *mex*. *mex* is a character size relative to default character size (like *cex*) and margin coordinates are measured in terms of characters of this size.

`mfg=c(i,j,m,n)` (P) Multiple figure parameters which give the row and column number of the current multiple figure and the number of rows and columns in the current array of multiple

figures.

`mfrow=c(m,n)` (P) Subsequent figures will be drawn row-by-row in an m by n matrix on the page.

`mfcol=c(m,n)` (P) Subsequent figures will be drawn column-by-column in an m by n matrix on the page.

`mgp=c(x1,x2,x3)` margin parameters which give the margin coordinate for the axis title, axis labels, and axis line.

`mkh=x` height in inches of mark symbols drawn by the `mark=` argument to `points`.

`new=1` (P) If TRUE, the current plot is assumed to have no previous plotting on it. Any points, lines, or text will set `new` FALSE.

`oma=c(x1,x2,x3,x4)` (P) Specification of the outer margin coordinates in terms of text of size `mex`. `oma` provides the maximum value for outer margin coordinates on each of the four sides of the multiple figure region. `oma` causes recreation of the current figure within the confines of the newly specified outer margins.

`omd=c(x1,x2,y1,y2)` (P) The region within the outer margins (which is to be used by multiple figure arrays) is specified by `omd` as a fraction of the entire device.

`omi=c(x1,x2,x3,x4)` (P) Size of outer margins in inches.

`pch=ch` the character to be used for plotting points. If `pch` is a period, a centered plotting dot is used.

`pin=c(w,h)` (P) Width and height of plot, measured in inches.

`plt=c(x1,x2,y1,y2)` (P) The coordinates of the plot region measured as a fraction of the figure region.

`pty=ch` (P) The type of plotting region currently in effect. Values: "s" generates a square plotting region; "m" generates a maximal size plotting region, which, with the margins, completely fills the figure region.

`smo=x` smoothness of circles and other curves. `smo` is the number of rasters that the straight-line approximation to the curve is allowed to differ from the exact position of the curve. Large values produce more crude approximations to curves, but allows the curves to be drawn with fewer line segments and hence speeds up output. The minimum number of line segments that will be used for a circle is 8, regardless of `smo`.

`srt=x` string rotation in degrees measured counterclockwise from horizontal. When specified, sets `crt` to same value.

`sub=s` (H) Sub-title, to be placed below the X-axis label in standard sized characters.

`tck=x` the length of tick marks as a fraction of the smaller of the width or height of the plotting region. If `tck` is negative, ticks are drawn outside of the plot region. If `tck` = 1, grid lines are drawn.

`type=ch` (H) Type of plot desired. Values are "p" for points, "l" for lines, "b" for both points and lines, "n" for no plotting, and "h" for high-density vertical line plot.

`usr=c(x1,x2,y1,y2)` (P) User coordinate limits (min and max) on X and Y axes.

`[xy]lab=s` (H) Label for plotting below the x- or y-axis.

`[xy]axp=c(ul,uh,n)` axis parameters giving coordinates of lower tick mark *ul*, upper tick mark *uh*, and number of intervals *n* within the range from *ul* to *uh*.

`[xy]axs=ch` style of axis interval calculation. The styles "s" and "e" set up standard and extended axes, where numeric axis labels are more extreme than any data values. In addition, extended axes may be extended another character width so that no data points lie very near the axis limit. Style "i" creates an axis labelled internal to the data values. This style wastes no space, yet still gives pretty labels. Style "d" is a direct axis, and axis parameters will not be changed by further high-level plotting routines. This is used to "lock-in" an axis from one plot to the next. Default is "e".

`[xy]axt=ch` axis type. Type "n" (null) can be used to cause an axis to be set up by a high-level routine, but then not plotted.

`[xy]lim=c(x1,x2)` (H) Approximate minimum and maximum values to be put on x- or y-axis. These values are automatically rounded to make them "pretty" for axis labelling.

`xpd=1` logical value controlling clipping. Values: FALSE = no points or lines may be drawn outside of the plot region. TRUE = points, lines, and text may be plotted outside of the plot region as long as they are inside the figure region.

pardump: Dump of Graphical Parameters

`pardump(arg1, arg2, ...)`

ARGUMENTS:

`argi:` names of graphical parameters to be dumped (see *par* for a list of the names).

EFFECT:

Prints a dump of the graphical parameters currently in effect. If arguments are given to *pardump*, only the parameters with the corresponding names will be printed. If no arguments are given, all parameters are printed. Parameters are listed alphabetically. With no arguments given, *pardump* will produce about a page of output.

pbeta: (see *beta*)

pcauchy: (see *cauchy*)

pchisq: (see *chisq*)

persp: 3-dimensional Perspective Plots

`persp(z, eye, ar)`

ARGUMENTS:

- z:** matrix of heights given over a regularly spaced grid of x and y values, i.e., $z[i,j]$ is the height at $x[i],y[j]$. Although x and y are not input to *persp*, the algorithm plots as if x and y are in increasing order and equally spaced from -1 to 1.
- eye:** vector giving the x,y,z coordinates for the viewpoint. Default is (-6,-8,5). Since the implied x,y grid ranges over (-1,1), the x,y coordinates of *eye* should not both be in the range (-1,1).
- ar:** aspect ratio of the actual x,y grid, i.e., $(x_{\max}-x_{\min})/(y_{\max}-y_{\min})$. Default 1.

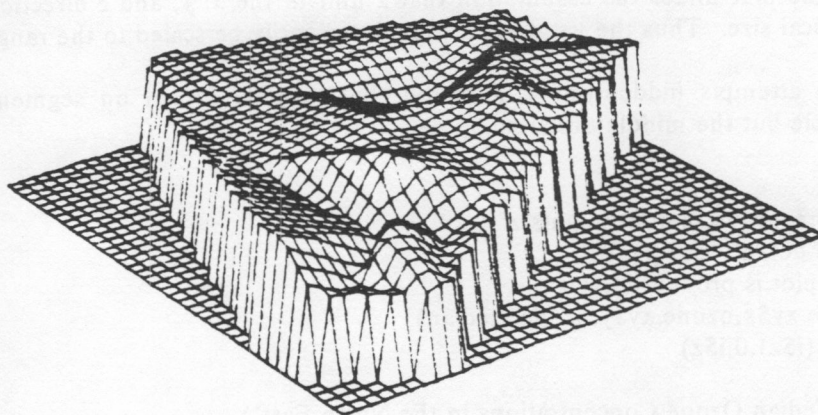
persp sets up the plot under the assumption that a unit in the x , y , and z directions represents the same physical size. Thus the values in z should ordinarily be scaled to the range (0,1).

The algorithm attempts hidden-line elimination, but may be fooled on segments with both endpoints visible but the middle obscured.

EXAMPLES:

```
persp(z) #perspective plot of heights z
        #from default viewpoint
#the example plot is produced by:
i←interp(ozone.xy$x,ozone.xy$y,ozone.median)
iSz←ifelse(NA(iSz),0,iSz)
persp(iSz/200)
title(main="Median Ozone Concentrations in the North East")
```

Median Ozone Concentrations in North East



pf: (see f)

pgamma: (see gammadist)

photo prism stare: Deferred Graphics Devices

photo
prism
stare

These functions are for deferred graphics devices, specifically the photographic output from the FR80 microfilm recorder, color plotting from the Xerox 6500 color plotter, and medium-quality output from the stare3 plotter.

When graphic input (*identify*, *rdpen*) is requested, the prompt *x,y:* appears on the users terminal, and the user should type the (x,y) coordinate pair of the point to be input. To terminate the input, hit carriage return. Graphic input is not allowed in batch mode.

Character sizes may be changed to any multiple of .5. In addition, *photo* has numerous other character sizes. On *prism*, color can be changed to any of the values 1 to 8, indicating colors black, red, blue, green, purple, yellow, magenta, and white. Characters can be rotated to any orientation on *photo*, and to 0 or 90 degrees on *stare* and *prism*. Different line styles (1, 2, ...)

are available.

A device must be specified before any graphics functions can be used.

pie: Pie Charts

`pie(x, label, size, inner, outer)`

ARGUMENTS:

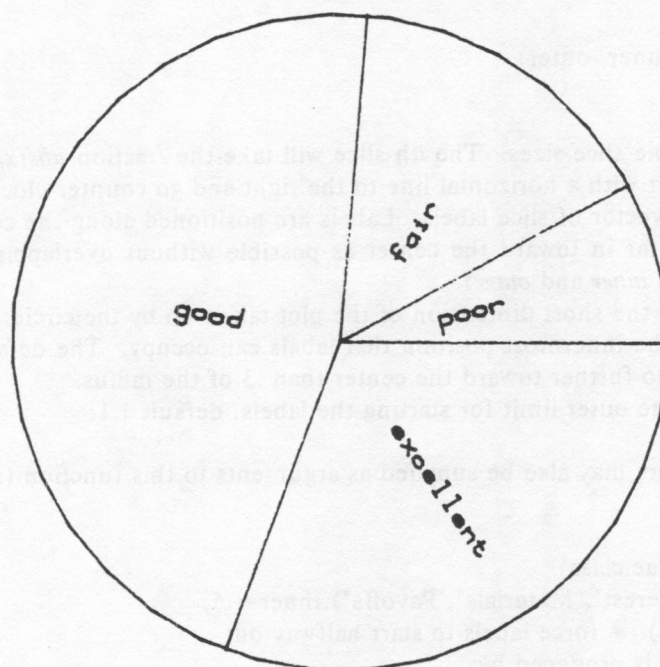
- x:** vector of relative pie slice sizes. The *i*th slice will take the fraction $\text{abs}(x[i])/\text{sum}(\text{abs}(x))$ of the pie. The slices start with a horizontal line to the right and go counter-clockwise.
- label:** optional character vector of slice labels. Labels are positioned along the center-line of the slice, and are shifted as far in toward the center as possible without overlapping into adjacent slices (but see arguments *inner* and *outer*).
- size:** optional fraction of the short dimension of the plot taken up by the circle. Default .75.
- inner:** optional fraction, the innermost position that labels can occupy. The default value of .3 means that labels can go no further toward the center than .3 of the radius.
- outer:** optional fraction, the outer limit for starting the labels, default 1.1.

Graphical parameters may also be supplied as arguments to this function (see *par*).

EXAMPLES:

```
pie(revenues,revenue.class)
pie(expenses,c("Interest","Materials","Payoffs"),inner=.5,
  outer=.5,size=1) # force labels to start halfway out
# the example plot is produced by:
datatel—?col(telsam.respons,sum)
pie(datatel,telsam.collab)
title(main="Response to Quality of Service Questions
concerning Telephone Service")
```

Response to Quality of Service Questions concerning Telephone Service



plclust: Plot Trees From Hierarchical Clustering

plclust(tree, hang, unit, level, hmin, square, label, plot)

ARGUMENTS:

- tree: a hierarchical clustering tree, of the form returned by function *hclust*.
- hang: the fraction of the height of the plot that any individual node will hang below the cluster that it joins. A value of -1 will cause all individuals to start at y-value 0. Default 0.1.
- unit: logical flag. If TRUE, the heights of the merges will be ignored and instead merge *i* will occur at height *i*. Useful for spreading out the tree to see the sequence of merges. Default is FALSE.
- level: logical flag. If TRUE, plotted tree will be "leveled", where merges in different subtrees are arbitrarily assigned the same height in order to compress the vertical scale. Particularly useful with *unit=TRUE*. Default is FALSE.
- hmin: optional minimum height at which merges will take place. Can be used to get rid of irrelevant detail at low levels. Default 0.
- square: logical flag. If TRUE (default), the tree is plotted with "U" shaped branches, if FALSE, it has "V" shaped branches.
- label: optional character vector of labels for the leaves of the tree. If omitted, leaves will be labelled by number. To omit labels entirely, use *label=FALSE*.
- plot: logical flag. If TRUE (default), plotting takes place. If FALSE, no plotting is done (useful for returned value).

Graphical parameters may also be supplied as arguments to this function (see *par*).

VALUE:

if *plot* is FALSE, a structure containing the coordinates of the leaves of the tree and the interior nodes of the tree.

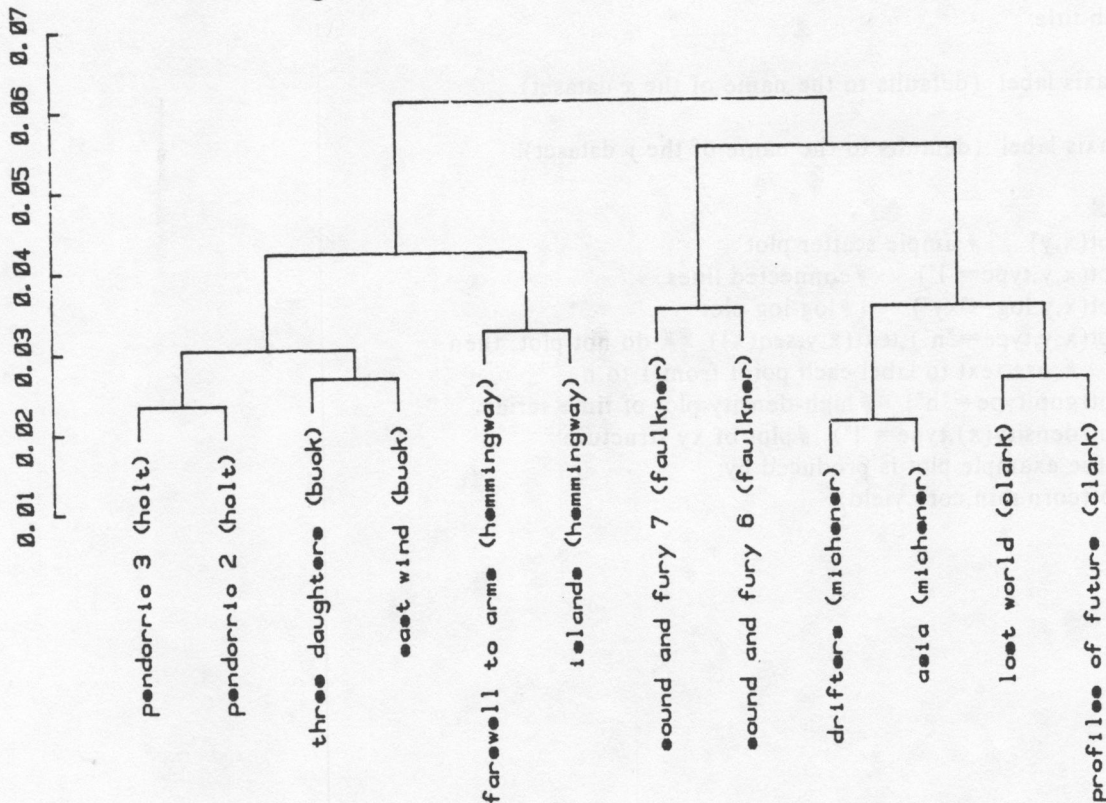
x,y: *x* and *y* coordinates of the leaves of the tree, i.e., *x[i],y[i]* gives the coordinates of the leaf corresponding to the *i*th individual.

xn,yn: *x* and *y* coordinates of the interior nodes of the tree, i.e., *xn[i],yn[i]* gives the coordinates of the node representing the *i*th merge.

EXAMPLES:

```
plclust(hclust(distances))
plclust(tree,label=FALSE) # plot without labels
xy←plclust(tree,plot=FALSE) # no plot, save structure
  identify(xy) #allow user to point at leaf
    #and have it identified
# the example plot is produced by:
$T←?row($author.count,sum)
$T←?rsweep(author.count,$T,/)
par(mar=c(18,4,4,1))
plclust(hclust(dist($T)),label=author.rowlab)
title("Clustering of Books Based on Letter Frequency")
```

Clustering of Books Based on Letter Frequency



plnorm: (see **lnorm**)

logis: (see **logis**)

plot: Scatter Plots

`plot(x, y)`

ARGUMENTS:

x,y: coordinates of points on the scatter plot. A time series or structure containing components named *x* and *y* may also be given for *x*. Missing values (NAs) are allowed.

Graphical parameters may also be supplied as arguments to this function (see *par*).

Graphical parameters that are particularly useful with *plot*:

type= where values of "p", "l", "b", "n", and "h" produce points, lines, both, nothing, and high-density lines.

log= where values of "x", "y", or "xy" specify which axes are to be on a logarithmic scale.

main= main title

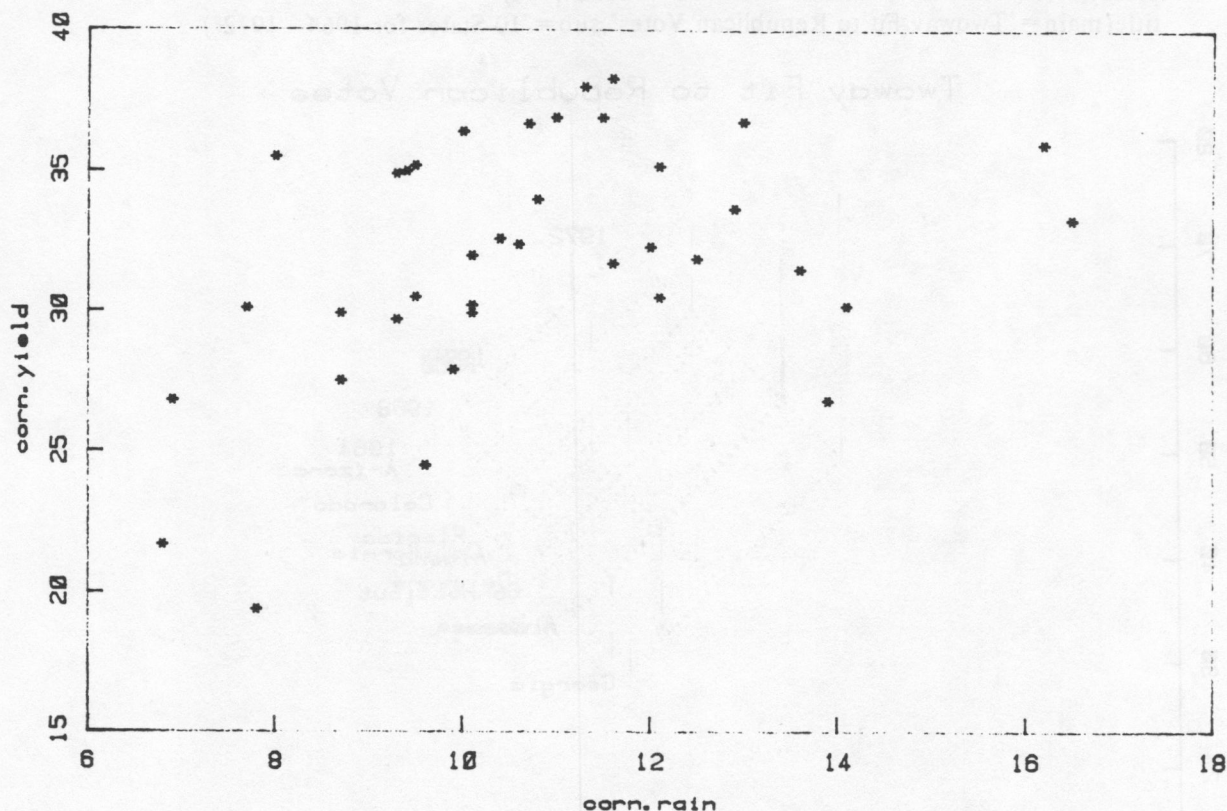
sub= sub-title

xlab= x axis label (defaults to the name of the *x* dataset).

ylab= y axis label (defaults to the name of the *y* dataset).

EXAMPLES:

```
plot(x,y)      #simple scatter plot
plot(x,y,type="l")  #connected lines
plot(x,y,log="xy")  #log-log plot
plot(x,y,type="n");text(x,y,seq(x))  # do not plot, then
    # use text to label each point from 1 to n
plot(gnp,type="h")  # high-density plot of time series
plot(density(x),type="l")  #plot of xy structure
# the example plot is produced by:
plot(corn.rain,corn.yield)
```

plotfit: Two-way Plot of Fit

plotfit(fit, w, c, rowlab, collab, grid)

ARGUMENTS:

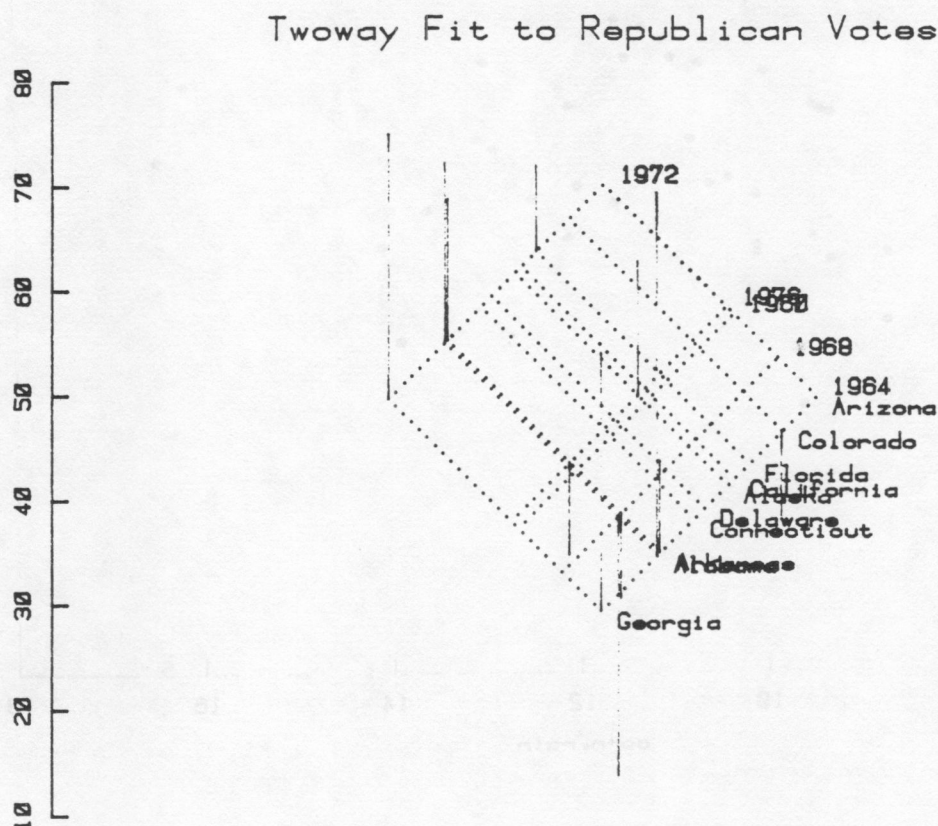
- fit: structure with components *row*, *col*, *grand*, and *resid*, reflecting a two-way fit to a matrix. See, for example, the output of function *twoway*.
- w: interaction term, i.e., the coefficient of the row*col interaction. Default 0. The residuals in *fit* should NOT reflect this term, i.e., $data[i,j] = grand + row[i] + col[j] + resid[i,j]$.
- c: residuals larger in magnitude than *c* will be displayed. If $c < 0$, no residuals will be displayed; if $c = 0$, all residuals will be displayed. Default -1.
- rowlab: character vector giving labels for the rows of the matrix. Defaults to "Row i". To omit labels, use *rowlab*="".
- collab: character vector giving labels for the columns of the matrix. Defaults to "Col i". To omit labels, use *collab*="".
- grid: should grid of fitted values be drawn? Default TRUE.

Graphical parameters may also be supplied as arguments to this function (see *par*).

EXAMPLES:

```
plotfit(twoway(datamat))
# the example plot is produced by:
vy←votes.year[27:31] #get last five election years
vr←twoway(votes.repub[1:10,27:31]) #first 10 states, last 5 years
```

```
plotfit(vr,c=8,rowlab=state.name,collab=encode(vy))
title(main="Twoway Fit to Republican Votes",sub="10 States for 1964 - 1972")
```



10 States for 1964 - 1972

pmax pmin: Parallel Maximum or Minimum

```
pmax(arg1, arg2, ...)
pmin(arg1, arg2, ...)
```

ARGUMENTS:

argi: numeric structure.

VALUE:

vector whose first element is the maximum (*pmax*) or minimum (*pmin*) of the first elements of the arguments, and similarly for the second element, etc. The length of the vector is the length of the longest argument. Shorter vectors are reused cyclically. Missing values (NA) are allowed; if an element of any of the arguments is NA, the corresponding element of the result is also NA.

EXAMPLES:

```
pmax(x,y,5) # largest of x[i],y[i] and 5
```

Note the difference between *pmax*, *pmin* and *max*, *min*. The latter give the single element which is the max or min of all the arguments. See also *range*.

pmin: (see **pmax**)

pnorm: (see **norm**)

points: (see **lines**)

ppoints: Plotting Points for Q-Q Plots

`ppoints(n)`

ARGUMENTS:

n: sample size for which plotting points desired (if *n* has only 1 value) or data against which plot is to be made.

VALUE:

the vector of probabilities, *p*, such that *qdist(p)* plotted against *sort(y)* gives a probability (Q-Q) plot of *y* against the distribution of which *qdist* is the quantile function. Computes $p[i] = (i-.5)/n$ if $n > 10$ or $(i-.375)/(n+.25)$ otherwise.

EXAMPLES:

```
plot(qlnorm(ppoints(y)),sort(y)) #log normal q-q plot
```

prcomp: Principal Component Analysis

`prcomp(x, retx)`

ARGUMENTS:

x: data matrix to be decomposed. Principal component analysis defines a rotation of the variables (columns) of *x*. The first derived direction is chosen to maximize the standard deviation of the derived variable, the second to maximize the standard deviation among directions uncorrelated with the first, etc.

retx: logical, if TRUE (the default) the rotated version of the data matrix is returned. Setting to FALSE just saves space in the returned data structure.

VALUE:

structure describing the principal component analysis:

sdev: standard deviations of the derived variables.

rotation: orthogonal matrix describing the rotation. The first column is the linear combination of columns of *x* defining the first principal component, etc. May have fewer columns than *x*.

x: if *retx* was TRUE, the rotated version of *x*; i.e., the first column is the *nrow(x)* values for the first derived variable, etc. May have fewer columns than *x*.

The analysis will work even if $nrow(x) < ncol(x)$, but in this case only *nrow(x)* variables will be derived, and the returned *x* will have only *nrow(x)* columns. In general, if any of the derived variables has zero standard deviation, that variable is dropped from the returned result.

precedence

Precedence determines the order of evaluation of a non-parenthesized expression. Operations of a higher precedence are evaluated before any operations of lower precedence. Equal precedence operators are evaluated left-to-right (except assignment and exponentiation which are right-to-left).

\$	component select	HIGH
%x	special operator	
-	unary minus	
:	sequence operator	
~ **	exponentiation	
* /	mult/div	
+ -	add/sub	
< > <= >= == !=	logical	
!	not	
&	and/or	
← _ →	assignment	LOW

prefix: Set Prefix for Dataset Names

prefix(name)

ARGUMENTS:

name: character string, the characters in which must be legal in dataset names; (i.e., letters, digits and "."). The function then has the side effect that this string is automatically prepended to all dataset names used in subsequent expressions. The prefix may be overridden by giving a fully-qualified dataset name and starting the name with "\$". If the argument is omitted, the prefix is made empty (also the initial state).

VALUE:

the previous prefix, so that one can store and restore the prefix (useful in macros). The result of *prefix* is not automatically printed.

It is recommended that *prefix* be used to distinguish datasets generated during a particular analysis. Thus, for example, if we want to analyse some data concerning wages, the prefix may be set to "wages." with the result that commonly used names, such as "x", "y", "label", etc. do not conflict with datasets of the same name, used in analysis of other data. Slashes may also be used in prefixes (and in dataset names in general), but note that these specify subdirectories under the directory containing the data. The construction of subdirectories corresponding to prefixes is a generally good idea, as it avoids the UNIX restriction of dataset names to 14 characters.

EXAMPLES:

```
oldp←prefix("wages.") #set prefix, store old one
print(x,y,$z) #print wages.x wages.y and z
print( prefix() ) #delete prefix and print its previous value
```


pretty: Return Vector of Prettied Values

```
pretty(x, nint)
```

ARGUMENTS:

x: vector of data; prettied values will cover range of *x*.
nint: optional, approximate number of intervals desired. Default 5.

VALUE:

vector of (ordered) values, defining approximately *nint* intervals covering the range of *x*. The individual values will differ by 1, 2 or 5 times a power of 10.

EXAMPLES:

```
pretty(mydata,10)
```

print: Print Data

```
print(arg1, arg2, ..., rowlab=, collab=, max=)
```

ARGUMENTS:

argi: any structure. Missing values (NAs) are allowed.
rowlab=: character vector of labels for the rows of a matrix.
collab=: character vector of labels for the columns of a matrix.
max=: limit on number of values printed in a single vector or component, default is value from *max* parameter of *options*. Use a small value, say *max=10*, to print just the structure information about large data sets. See also *compname* for this purpose.

VALUE:

structure composed of all *args*.

The function is invoked automatically if a command line evaluates to a result which is not assigned a name.

If *argi* is given in *name=value* form, *name* is printed to label the value, and is used for the component name in the returned structure.

See *options* for control of line width and spacing.

PRINTDOC: (see PROMPT)**printer: Line Printer Graphics**

```
printer  
printer(width,length)
```

ARGUMENTS:

width: number of characters for width of printer plots. Default is value of *width* as set in *options* function (initially 80). Maximum width is 133.
length: desired number of lines for length of plot. Default is value of *length* as set in *options* function (initially 56). Maximum length is 100.

The *printer* device is applicable to any typewriter-like terminal. The quality is not equal to that of true graphics devices, but may be sufficient for exploratory data analysis.

The size of the printer plot is determined by the terminal width and length in effect at the time the function *printer* is invoked.

Each plot is stored in an internal buffer. This enables commands to be given to add to the existing plot. When the next plot is started, the previous plot is printed. The function *show* can be used to print the current plot. After using *show*, the plot can be further augmented. *printer* and *show* can also be used to preview plots that are to be produced by the deferred graphics facility *defer*. Use *show* to view the picture, then decide if it should be saved.

Character size and orientation cannot change. Different line types are available, but the resolution of the printer is generally too coarse to make much distinction.

Any graphic input (*identify*, *rdpen*) done on the printer will prompt for x and y coordinates. Type in the desired coordinates, or hit carriage return to terminate the graphic input.

A device must be specified before any graphics functions can be used.

EXAMPLES:

```
options(width=120) # for terminal with long line
printer; plot(x,y)
title("a title to be added to the plot")
show # we want to see the plot now
```

prism: (see **photo**)

prod sum: Products and Sums

```
prod(x)
sum(x)
```

ARGUMENTS:

x: numeric structure.

VALUE:

the sum (product) of all elements of x.

PROGRAM: Creating Stand-alone FORTRAN, RATFOR, or C programs

To create stand-alone programs which use the S libraries of algorithms and/or are written with the S facilities (as described in sections 6, 7 and 8 of the manual), use the command:

```
S PROGRAM [options] name files ...
S MAKE name
```

ARGUMENTS:

options: optional flag, where the value -g indicates a graphics program that should search the graphics libraries.

name: name of the stand-alone program to be produced.
files: names of files on which the program depends (in addition to standard S algorithm libraries), typically source files (which must end in ".r", ".f", or ".c") or library files.

The *PROGRAM* command is used once to specify the name of the stand-alone program and the names of source files of programs (including a main program) needed to produce *name*.

The *MAKE* command will ensure that the executable program is loaded with up-to-date object versions of the files on which it depends.

PROMPT: Utilities for Documenting S Functions

S PROMPT file.i ...
S NEWDOC file.d ...
S EDITDOC [-f] name ...
S PRINTDOC [-options] [name ...]

These routines provide mechanisms to help maintain documentation for chapters of user-written functions, macros or datasets.

The *PROMPT* utility takes a list of files which contain interface routines (these files have names ending in ".i"). For each "*file.i*", *PROMPT* produces an outline of documentation on "*file.d*". The system macro *?prompt* provides a similar facility for constructing the outline of documentation for datasets and macros.

Each of the resulting ".d" files should be edited with a text editor. Within each file are a number of lines which include the character "~" and a brief description of what should be documented at that point. These files use a number of *documentation macros* to control the formatting of the printed documentation. For example, the lines beginning ".AG" introduce function arguments; ".RC" returned components; ".FN" the names of functions documented in this file; ".TL" introduces the title; ".CS" introduces the calling sequence; ".EX" introduces lines of examples. The documentation macro ".PP" is used to start a paragraph, and results in a blank line in the printed documentation. ".IP topic" creates an indented paragraph that is labelled by *topic*. These last two macros can be used to break the documentation into appropriate paragraphs.

Once the documentation outline files have been edited, they should be installed into the chapter documentation directory by means of the *NEWDOC* utility. *NEWDOC* will automatically install macro and dataset documentation into the database documentation. Once this installation is done, the documentation will be accessible via the *help* function as long as the chapter or database are being searched.

Once the documentation is installed by *NEWDOC*, the ".d" file can be thrown away. When it is necessary to change the documentation after it has been installed, the utility *EDITDOC* can be used. When given a list of documentation names, *EDITDOC* looks through the database and chapter documentation for the named documents. The documentation is then extracted and placed on file *name.d*. This file can be edited, and then re-installed by means of *NEWDOC*. If a function has the same name as a macro or dataset, *EDITDOC* can be told to retrieve the function documentation by means of the optional "-f" flag.

The utility *PRINTDOC* is used to construct a neatly-formatted listing of documentation. By default, all macro and dataset documentation is printed on the user's terminal. If a list of

names is given, the documentation corresponding to those names is printed. The *options* list is a set of characters preceded by a dash, and controls the choice of output device and the listing of function documentation. If *options* contains the character "f", documentation for functions (rather than macros and datasets) will be printed. The character "t" indicates that *troff* should be used to produce final documentation on the phototypesetter. The character "o" indicates that documentation is to be done on an offline printer, not the terminal.

EXAMPLES:

```
S PROMPT myfun.i
... edit the file myfun.d ...
S NEWDOC myfun.d
S PRINTDOC -ft      # typesets all function documentation
```

pt: (see **tdist**)

punif: (see **unif**)

q: Terminating Execution

q

Causes termination of execution and returns to the operating system. If deferred graphics is on at the time, the last frame will be saved (see *defer*).

qbeta: (see **beta**)

qcauchy: (see **cauchy**)

qchisq: (see **chisq**)

qf: (see **f**)

qgamma: (see **gammadist**)

qlnorm: (see **lnorm**)

qlogis: (see **logis**)

qnorm: (see **norm**)

qqgamma: (see **qqplot**)

qqnorm: (see **qqplot**)

qqplot qqnorm qqgamma: Quantile-Quantile Plots

```
qqplot(x, y, plot)
qqnorm(xy, datax, plot)
qqgamma(xy, eta, datax, plot)
```

ARGUMENTS:

x,y: vectors (not necessarily of the same length). Each is taken as a sample, for the *x* and *y* axis values of an empirical probability plot.

xy: vector of data for a normal or gamma probability plot.

datax: logical flag, if TRUE, data goes on the *x* axis, if FALSE (default) data goes on the *y* axis.

eta: shape parameter to be used in the gamma plot. Must be supplied.

plot: logical flag. If *plot* is FALSE, these routines all return a structure with components *x* and *y* which give the coordinates of the points that would have been plotted. Default is TRUE.

Graphical parameters may also be supplied as arguments to this function (see *par*).

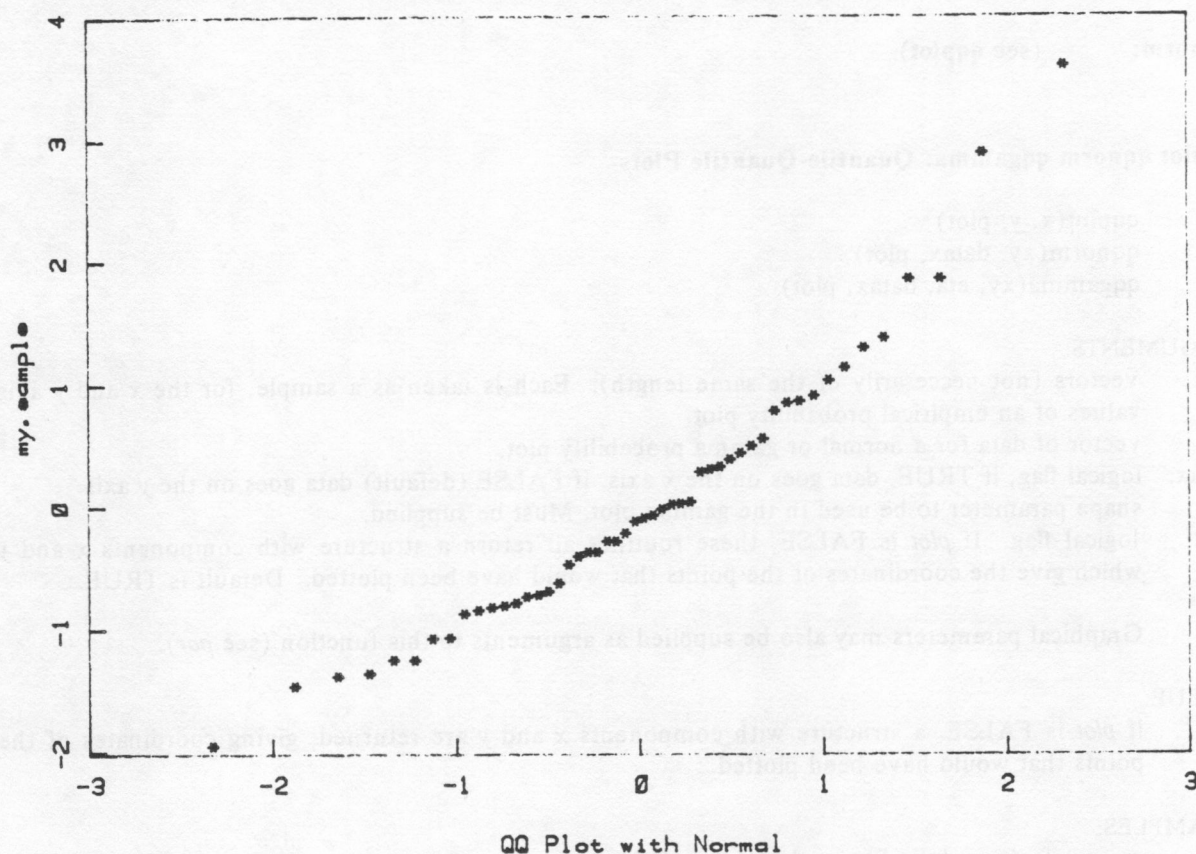
VALUE:

if *plot* is FALSE, a structure with components *x* and *y* are returned, giving coordinates of the points that would have been plotted.

EXAMPLES:

```
zz←qqplot(x,y,plot=F)    #save x and y coords of empirical qq
plot(zz)                 #plot it
abline(rbiwt(zz$x,zz$y)) #fit robust line and draw it
# the example plot is produced by:
my.sample←rt(50,5)
lab←"50 samples from a t distribution with 5 d. f."
qqnorm(my.sample,main=lab,sub="QQ Plot with Normal")
```

50 samples from a t distribution with 5 d.f.



qt: (see **tdist**)

qunif: (see **unif**)

range: Range of Data (minimum,maximum)

`range(arg1, arg2, ...)`

ARGUMENTS:

argi: numeric structure. Missing values (NAs) are allowed.

VALUE:

vector of two elements, the first the minimum of all the elements of all arguments; the second the maximum. This function is useful as an argument to plotting when it is desired to specify the limits for the x- or y-axes. (See the example.)

EXAMPLES:

`plot(x,y,ylim=range(y,0,1))` # force range to include (0,1)

rank: Ranks of Data

rank(x)

ARGUMENTS:

x: numeric structure.

VALUE:

the ranks; i.e., the i -th value is the rank of $x[i]$. In case of ties, the average rank is returned.

RATFOR: (see **PROGRAM**)

rbeta: (see **beta**)

rbind: (see **cbind**)

rbiwt: Robust Simple Regression by Biweight

rbiwt(x, y)

rbiwt(x, y, start, k, tol, iter)

ARGUMENTS:

x: vector of observations on independent variable.

y: vector of observations on dependent variable.

start: vector giving starting values of intercept and slope. Default, use least-squares *start*.

k: biweight scale parameter, default 6.

tol: convergence criterion. Default .001

iter: maximum number of iterations. Default 20.

VALUE:

structure containing components *coef*, *resid*, and *wt*.

coef: vector giving intercept and slope.

resid: vector like *y* giving residuals from fit.

wt: vector giving weights used in final weighted least-squares step.

Does not fit data to more than 1 independent variable (see *rreg*).

REFERENCE:

Coleman, D., Holland, P., Kaden, N., Klema, V., and Peters, S. C. (1980). "A system of subroutines for iteratively re-weighted least-squares computations", *ACM Trans. Math. Soft.*, vol. 6, 327-336.

EXAMPLES:

plot(x,y); abline(rbiwt(x,y)) #add line to plot

rcauchy: (see **cauchy**)

rchisq: (see **chisq**)

rdpen: Get Coordinates from Plot

rdpen(n)

ARGUMENTS:

n: the maximum number of points to identify. Default is 500.

VALUE:

structure containing vector components *x* and *y* which give co-ordinates for each point. The length of these vectors is at most *n*, but can be shorter if the user terminates graphic input after fewer than *n* points are given.

See the individual device documentation for the procedure to use in identifying points.

EXAMPLES:

```
text(rdpen(1),"outlier") #user points at outlier which is then labelled
lines(rdpen())           # input a number of points, connect with line
```

read: Read Data from a File

read(file, length, mode, skip, print)

ARGUMENTS:

file: character string, naming the file to read. If the file name is omitted, reading is from the standard input (terminal).

length: the maximum number of values to read, default 2000.

mode: the mode of the data, default **REAL**.

skip: number of lines of the file to skip before starting to read. Default 0.

print: logical flag, should number of items read be printed? Default **TRUE**.

VALUE:

vector comprised of the data read from the file. If fewer than *length* values are read before end-of-file, the length of the result is the number of values read.

Also prints the number of values read, and warns if end-of-file was not reached.

Character strings read (*mode=CHAR*) can contain internal blanks only if enclosed by quotes.

EXAMPLES:

```
tst ← read("testdata")      # normal usage
```


reg regprt: Regression

```
reg(x, y, wt, int, names, q)
regprt(regstr, names)
```

The functions *reg* and *regprt* are components of the *regress* function. *regress(...)* is equivalent to *regprt(reg(...))*. *reg* is identical to *regress* except it does not print (hence it is useful if the regression is used for intermediate results). *regprt* takes the structure produced by *reg* and prints a summary of the regression statistics. *reg* and *regprt* can take a vector of names to augment the printing of the regression summary.

See *regress* for details about arguments.

regress: Regression

```
regress(x, y, wt, int, print, names, q)
```

ARGUMENTS:

- x: x matrix for fitting $Y = Xb + e$ with variables in columns, observations across rows. Should not contain column of 1's, (see argument *int*). Number of rows of *x* should equal the number of data values in *y*. There should be fewer columns than rows.
- y: numeric vector with as many observations as the number of rows of *x*.
- wt: numeric vector of weights for weighted regression. Should be the same length as *y*.
- int: flag, if TRUE (the default) a constant (intercept) term is included in the regression.
- print: flag, if TRUE (the default), the regression results are printed. Compare functions *reg* and *regprt*. For *reg*, argument *print* is FALSE by default.
- names: optional character vector giving the names of the variables in the matrix *x*. By default, names are "Var 1", "Var 2", etc.
- q: logical flag, if TRUE, the *q* matrix from an orthogonal decomposition of *x* is returned. Default FALSE.

VALUE:

structure with the following components:

- coef: vector of coefficients with (optional) constant term first.
- resid: structure like *y* containing residuals.
- r,corth: components of orthogonal decomposition of *x* matrix.
- names: names of variables (for later use by *regprt*).
- int: records whether intercept was used in this regression.
- q: matrix from orthogonal decomposition if *q* argument was TRUE.
- sqrtw: vector of the square-roots of the input vector *wt* (only if *wt* was given).

EXAMPLES:

```
regress(cbind(a,b,c),y) #regress y on a, b, and c with intercept
```

regsum: Regression Summaries

regsum(z)

ARGUMENTS:

z: regression structure, with components *coef*, *resid*, *r*, *corth*, *int* and (possibly) *sqrtw*. See documentation for *regress*.

VALUE:

structure with components summarizing the regression statistics.
names: vector of names for the *x* variables in the regression, either the names provided as arguments to *reg* or encoded names. If the regression has an intercept, "Intercept" is the first name.
table: matrix with 3 columns and a row for each coefficient, either the number of *x* variables or one more if an intercept was included. The three columns are coefficient value, standard error and t-statistic.
cor: correlation matrix of the coefficients.
cov: covariance matrix of the coefficients.
rsq: multiple R-square statistic (the fraction of the variance of *y* explained by the model).
rms: standard error of the residuals.
fval: the overall F-statistic value for the model.

EXAMPLES:

```
z ← reg(x,y) #do the regression
zsumy ← regsum(z) #get the summary
```

rep: Replicate Data Values

rep(x, times, length)

ARGUMENTS:

x: numeric structure.
times: how many times to replicate *x*. There are two ways to use *times*. If it is a single value, the whole of *x* is replicated that many times. If it is a vector of the same length as *x*, the result is a vector with *times*[1] replications of *x*[1], *times*[2] of *x*[2], etc.
length: the desired length of the result. This argument may be given instead of *times*, in which case *x* is replicated as much as needed to produce a result with *length* data values.

VALUE:

a vector of the same mode as *x* with the data values in *x* replicated according to the argument *times* or *length*.

EXAMPLES:

```
rep(0,100) # 100 zeroes
rep(1:10,10) # 10 repetitions of 1:10
rep(1:10,1:10) # 1, 2, 2, 3, 3, 3, ...
```

replace:(see **append**)

replot: Plot Deferred Graphics File

```
replot(file)
```

ARGUMENTS:

file: optional name of deferred graphics file, default "sgraph".

The contents of *file* are plotted on the current graphics device from first frame to last. There is no provision for plotting single frames.

restore: Restore Datasets from Dump File

```
restore(file, pos, print)
```

ARGUMENTS:

file: character string giving the name of a file onto which the *dump* function dumped datasets.

pos: the position in the database search list of the database onto which the restore should put the datasets. Default 1 (the working database).

print: logical flag; when TRUE, dataset names are printed as they are put on database. Default FALSE.

Typically, the pair of functions *dump* and *restore* are used to move databases from one computer system to another. An entire database may be dumped, the resulting file transported to a file on another system, and the database restored.

NOTE:

The functions *dump* and *restore* ignore the current prefix.

EXAMPLES:

```
restore("dumpfile",2) # restore contents of dumpfile onto save db
```

rev: Reverse the Order of Elements in a Vector

```
rev(x)
```

ARGUMENTS:

x: vector. Missing values (NAs) are allowed.

VALUE:

vector like *x* but with the order of data values reversed (the last value in *x* is the first value in the result).

EXAMPLES:

```
rev(sort(t)) # sort t in descending order
```

rf: (see f)

rgamma: (see **gammadist**)

rlnorm: (see **lnorm**)

rlogis: (see **logis**)

rm: Remove Datasets from a Database

```
rm(arg1, arg2, ..., list =, print =, pos =, value =)
```

ARGUMENTS:

- argi:** datasets to be removed. Note that the arguments must actually correspond to datasets on a database; i.e., the argument is just a data-set name. Compare argument *list*.
- list=:** character vector of the names of datasets to be removed. The advantage over explicitly naming the datasets as above is that the datasets need not be accessed. Typically, the *list=* argument is used to remove many datasets at once, such as all the datasets under a prefix (see the example below).
- print=:** should names of removed datasets be printed? Default FALSE.
- pos=:** position in the database search list of the database from which the datasets should be removed. Default 1 (the working database).
- value=:** the value that the function should return. Useful in macros that generate temporary datasets that should be removed. This allows the *rm* function to be the last function in the macro, and to return a value.

VALUE:

the *value=* argument, if given.

EXAMPLES:

```
rm(x,y,z) #remove datasets x,y,z
rm(list=list("wages.*")) #remove anything on working database
                        # whose name starts with "wages."
rm(list=list("wages.*",pos=2),pos=2) #same for save database
rm(wages.x,pos=2) #remove wages.x from save database
rm($Tx,value=$Tx) #remove $Tx and return its value
```

rnorm: (see **norm**)

round: Rounding

```
round(x, dec)
```

ARGUMENTS:

- x:** numeric structure.
- dec:** number of decimal places desired. Default 0. *dec* can be negative, for rounding large numbers to 10, 100, etc.

VALUE:

structure like *x* with data rounded to the specified number of places.

EXAMPLES:

```
round(mydata,dec=2)  #round to 2 decimals
round(mydata,-1)    #round to nearest 10
```

row: (see col)

rreg: Robust Regression

```
rreg(x, y)          # simple form
rreg(x, y, w, int, init, method, wx, iter, k, acc, stop, conv)
```

ARGUMENTS:

- x:** matrix of independent variables for regression. Should not include a column of 1's for the intercept.
- y:** vector of dependent variable, to be regressed on *x*.
- w:** initial weights for robustness. *w* may be the weights computed from residuals in previous iterations of *rreg*. The argument *wx* should be used for weights that are to remain constant from iteration to iteration.
- int:** should intercept term be included in the regression? Default TRUE.
- init:** optional vector of initial coefficient values (normally the result of some other regression, e.g., *reg* or *lfit*). When omitted the initial value is computed as follows: if *wx* and/or *w* is supplied, it is the weighted least squares estimate, otherwise it is the ordinary least squares estimate.
- method:** choice of method (see below). Default is the converged Huber estimate followed by two iterations of Bisquare.
- wx:** optional weighting vector (for intrinsic weights, not the weights used in the iterative fit).
- iter:** maximum number of iterations; default 20.
- k:** constant in the weighting function. This constant is chosen to give the estimate a reasonable efficiency if the errors do come from a normal distribution. (See below for exact values.)
- acc:** convergence tolerance; default $10 \times \text{sqrt}(\text{machine precision})$.
- stop:** method of testing conversion. Values 1 (default), 2, 3, 4 use relative change in residuals, coefficients and weights and an orthogonality test of residuals to *x*.
- conv:** should component *conv* be returned as a result? Default FALSE.

VALUE:

structure with the following components:

- coef:** vector of coefficients in final fit.
- resid:** vector of final residuals.
- w:** vector of final weights in the iteration, excluding the influence of *wx*.
- int:** flag telling whether intercept was used.
- method:** name of robust weighting rule used.
- k:** value of *k* used for *method*.
- conv:** vector of the value of the convergence criterion at each iteration.

METHOD:

The routine uses iteratively reweighted least squares to approximate the robust fit, with residuals from the current fit passed through a weighting function to give weights for the next iteration. There are 8 possible weighting functions, all specified by character strings given as *method* (first character is enough): Andrews, Bisquare, Cauchy, Fair, Huber, Logistic, Talworth and Welsch. The default values of *k* are 1.339 for A, 6.2 for B, 2.385 for C, 1.4 for F, 1.345 for H, 1.205 for L, 2.795 for and 2.985 for W. Huber gives more least-squares-like fits usually; the proper choice of method, however, is still a research problem.

REFERENCE:

Coleman, D., Holland, P., Kaden, N., Klema, V., and Peters, S. C. (1980). "A system of subroutines for iteratively re-weighted least-squares computations", *ACM Trans. Math. Soft.*, vol. 6, 327-336.

rstab: (see **stab**)

rt: (see **tdist**)

runif: (see **unif**)

sample: Generate Random Samples or Permutations

sample(x, size, replace)

ARGUMENTS:

- x:** numeric or character vector of data (the population) to be sampled, or a scalar giving the size of the population. Missing values (NAs) are allowed.
- size:** sample size. Default is the same as the population size, and thus (with *replace=FALSE*) will generate a random permutation.
- replace:** logical flag, if TRUE, sampling will be done with replacement. Default FALSE.

VALUE:

if *x* is a population vector, the result is a sample from *x*; otherwise, the result is a set of integers between 1 and *x* giving the indices of the selected observations.

EXAMPLES:

```
sample(state.name,10) # pick 10 states at random
sample(1000000,75) # pick 75 numbers between 1 and one million
sample(50) # random permutation of numbers 1:50
```

sapply: Apply a Function to the Components of a Structure

sapply(x, fun, max=, arg1, arg2, ...)

ARGUMENTS:

- x:** hierarchical structure. An arbitrary S function will be applied to each component of *x*, and the result will become the corresponding component of the result of *sapply*.
- fun:** character string giving the name of the function.
- max=:** optional argument (only in the *name=expression* form) giving the maximum size for the result of any single application of the function. For example, if *fun* returned a result the same size as its argument, the maximum size of the components of *x* is the relevant number. Default 1000.
- argi:** other arguments to *fun*, if any.

VALUE:

structure whose first component is the result of *fun* applied to the first component of *x*, etc. If all the results are the same length, *sapply* returns a matrix with one column for each component. If all the results are scalars, a vector is returned.

EXAMPLES:

```
supply(x,"mean") # vector of means of components of x
supply(x,"sort") # sort the components
```

Note that the function *apply* can be used to perform similar operations on the sections of a matrix or array.

save: Save Data on Database

```
save(arg1, arg2, ..., pos=)
```

ARGUMENTS:

argi: structure to be saved on database. If *argi* is given in *name=value* form, the value is saved under the given name.

pos=: the position in the database search list of the database onto which the data should be saved. Default 2 (normally, the user's save database).

VALUE:

a structure composed of *arg1*, *arg2*,

save does not automatically print its result.

EXAMPLES:

```
print(save(a,b,y=1:10)) # saves a, b and y
                        # and prints their values
```

scale: Scale Columns of a Matrix

```
scale(x, center, scale)
```

ARGUMENTS:

x: matrix to be scaled. Missing values (NAs) are allowed.

center: control over the value subtracted from each column. If TRUE (the default), the mean is subtracted from each column. If given as a vector of length *ncol(x)*, this vector is used; i.e., *center[j]* is subtracted from column *j*. If FALSE, no centering is done.

scale: control over the value divided into each column to scale it. If TRUE (the default) each column (after centering) is divided by the square root of sum-of-squares over *n-1*, where *n* is the number of non-missing values. If given as a vector of length *ncol(x)*, column *j* is divided by *scale[j]*. If FALSE, no scaling is done.

VALUE:

matrix like *x* with optional centering and scaling.

The default values of *center* and *scale* produce columns with mean 0 and standard deviation 1.

EXAMPLES:

```
scale(x) # scale to correlation (0 mean, 1 std dev)
scale(x, center=apply(x,2,"median"),
      scale=FALSE) # remove column medians, do not scale
# see also sweep
```

search: Print Current Search List

```
search( which )
```

ARGUMENTS:

which: integer defining which search list to return: 1=chapters (for functions); 2=databases. Default 2.

VALUE:

character vector giving the list of database or chapter names currently being searched. See *chapter*, *attach* and *detach* for modifying the current search lists.

EXAMPLES:

```
search() #list of active databases
```

segments arrows: Plot Disconnected Line Segments or Arrows

```
segments(x1, y1, x2, y2)
arrows(x1, y1, x2, y2, size, open, rel)
```

ARGUMENTS:

x1,y1,x2,y2: co-ordinates of the end-points of the segments or arrows. Lines will be drawn from $(x1[i],y1[i])$ to $(x2[i],y2[i])$. Missing values (NAs) are allowed.

size: width of the arrowhead as a fraction of the length of the arrow if *rel* is TRUE. If *rel* is FALSE, *size* is arrowhead width in inches. Default 0.2.

open: logical, if TRUE the arrowhead is "v" shaped, if FALSE (default) it is diamond shaped.

rel: logical, should arrowhead be sized relative to the length of the arrow? Default FALSE.

Graphical parameters may also be supplied as arguments to this function (see *par*).

Note the distinction from *lines* which draws a curve (connected line segments). Any segments with any NAs as end-points will not be drawn.

\$ select: Component Selection

```
str$component
str$[icomp]
select(str, component)
```

ARGUMENTS:

str: an expression evaluating to a structure.

component: the name of a component of *str*. This can contain the minimum number of characters adequate to uniquely distinguish the component from other components in *str*.

icomp: an integer giving the position of a single component within *str*.

VALUE:

the selected component.

The \$ operator has very high precedence, and will be done before any other operators. Thus, if *str* is an expression returning a structure, it should ordinarily be parenthesized.

EXAMPLES:

```
regress(x,y)$coef
array2$Dim[1] # first element of Dim component of array2
for(i in seq(ncomp(z))) hist(z[i])
# histogram of each of the components of z
```

: seq: Sequences

```
from:to          # as operator
seq(from, to, by, length) # as function
```

ARGUMENTS:

from: starting value of sequence.
 to: ending value of sequence.
 by: spacing between successive values in the sequence.
 length: number of values in the sequence.

VALUE:

a numeric vector with values (*from*, *from*+*by*, *from*+2**by*, ... *to*). If arguments are omitted, an appropriate sequence is generated; for example, with only one argument a sequence from 1 to the value of the single argument is constructed. *from* may be larger or smaller than *to*. *by*, if specified, must have appropriate sign to generate a finite sequence. If only *from* is specified, and it is a vector with more than 1 value, a sequence from 1 to *len(from)* is generated.

When used as an operator, *:* has a very high precedence. Thus to create a sequence from 1 to *n-1*, parentheses are needed *1:(n-1)*.

EXAMPLES:

```
seq(5) # 1,2,3,4,5
1:5 # same thing
5:1 # 5,4,3,2,1
seq(0, 1, .01) # 0,.01,.02,.03,...,1.
seq(x) # 1, 2, ..., len(x)
seq(-3.14,3.14,len=100) # 100 values from -pi to pi
```

show: Show Current Printer Plot

```
show()
```

show provides a facility for viewing the current state of a printer plot. When *show* is invoked, the printer plot is produced. It can then be further augmented (with titles, lines, text), saved for deferred graphics, etc. If further graphics commands are used to augment a previously *shown* plot, the plot will be displayed again when any graphics functions advance to the next frame. If nothing has been added to a previously *shown* plot, the advance to the next frame will be silent.

EXAMPLES:

```
printer
plot(x,y) # scatter plot
show # preview it
title("an interesting plot") # add to it
```

sin: (see cos)

sink: Send S Output to a File

sink(file)

ARGUMENTS:

file: character string giving the name of a file.

The output that is ordinarily typed on the terminal is diverted to *file*. No output except prompt characters, printer plots, and error messages appears on the terminal. Diversion is ended when *sink* is given with no arguments.

smatrix: Print a Symbolic Matrix for Multivariate Data

smatrix(x, set, plength, nstrike, spread, scale, rowlab, collab, head, na)

ARGUMENTS:

- x: matrix of data. One symbol (possibly overstruck) will be printed for each element of *x*. One line of printing is produced for each row. Missing values (NAs) are allowed.
- set: optional character string. The number of characters per symbol is given by *nstrike* (below). If *nstrike*==1, the first character of *set* is the first symbol, etc. If *nstrike*>1, the first *nstrike* characters of *set* are overstruck to form the first symbol, the second *nstrike* characters to form the second symbol, etc. By default, if *nstrike*==1, the symbols are "+*". If *nstrike*==2, the default symbols are ".", "+", "*" and overstruck "\$" and "X". The equivalent value of *set* would be ". + * \$X".
- plength: number of lines to print per page (exclusive of running head). At the end of each page, a page eject is executed. The head string (see below) is printed at the top of each page. Default page length is 50.
- nstrike: number of lines superimposed to make the symbols. See the examples under *set* above. Default 1.
- spread: number of blanks (maximum) to put between the columns. Default 3.
- scale: logical, if TRUE, the columns of *x* will be scaled so that the minimum in each column is 0 and the maximum is 1. If FALSE, the assumption is that *x* has been scaled to the range 0 to 1 by some other procedure. Default TRUE.
- rowlab: optional character vector for labelling the rows. By default, labels are "1", "2", etc. and only every fifth label is plotted. If supplied, the label vector must have length equal to *nrow(x)*.
- collab: optional character vector for labelling the columns. By default, labels are "1", "2", etc. and only every fifth label is plotted. If supplied, the label vector must have length equal to *ncol(x)*.
- head: optional character string for a running head. This is plotted as a title at the top of each page. By default, the name of *x* is used. If there is more than one page, the page number is included in the title.
- na: character to be printed corresponding to missing values (NAs) in *x*. Default "N".

The symbols are chosen by dividing the range 0 to 1 into *ns* equal intervals, where *ns* is the number of characters in *set*, divided by *nstrike*.

EXAMPLES:

```
south — state.region == 2
smatrix(votes.repub[south,], rowlab=state.name[south],
        collab=encode(votes.year), head="Southern State Votes")
```


Southern State Votes

Alabama	N	N	N	+	+	+	+	+	+	.	+	+	.	+	+	+	*	.	.	.	+	.	+	+	*	.	*	+	
Arkansas	N	N	N	+	.	+	+	+	+	+	+	+	*	+	+	+	+	+	+	+	+	+	+	+	+	+	+	.	
Delaware	*	*	+	+	.	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Florida	N	N	N	N	+	*	*	*	*	N	+	.	.	.	.	+	+	*	+	+	+	+	+	+	+	+	+	*	
Georgia	N	N	N	.	.	.	+	+	+	+	+	+	.	+	.	+	.	.	.	.	.	.	.	.	.	.	+	+	
Kentucky	.	.	.	.	+	+	+	+	+	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
Louisiana	N	N	N	.	+	*	+	*	.	+	.	.	.	.	.	+	.	.	.	.	.	+	+	*	.	+	.	*	
Maryland	.	.	+	.	.	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
Mississippi	N	N	N	N	+	.	.	+	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	
North Carolina	N	N	N	+	+	*	*	*	*	*	*	*	*	.	*	*	*	*	+	+	+	+	+	+	+	+	+	+	
Oklahoma	N	N	N	N	N	N	N	N	N	N	N	N	*	*	+	*	*	*	+	*	*	*	*	*	*	*	*	*	
South Carolina	N	N	N	*	*	*	+	.	.	+	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	+	
Tennessee	N	N	N	*	.	+	*	*	*	*	*	*	*	+	*	*	*	*	+	*	*	*	*	*	*	*	*	+	
Texas	N	N	N	N	.	.	.	.	.	+	+	+	+	+	.	+	.	*	.	.	.	+	+	+	+	+	+	*	
Virginia	.	.	N	N	.	+	+	*	*	*	*	*	*	+	+	+	+	+	+	+	+	+	+	+	+	+	+	*	
West Virginia	N	N	*	*	.	+	*	*	*	*	*	*	*	+	*	*	*	*	*	*	*	*	*	*	*	*	+	.	

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
5	6	6	6	7	7	8	8	8	9	9	0	0	0	1	1	2	2	2	3	3	4	4	4	4	5	5	6	6
6	0	4	8	2	6	0	4	8	2	6	0	4	8	2	6	0	4	8	2	6	0	4	8	2	6	0	4	8

smooth: Non-linear Smoothing

smooth(x,t)

ARGUMENTS:

- x: vector (at least 5 points) to be smoothed.
t: logical flag, should twicing be done? Default TRUE. (Twicing is the process of smoothing, computing the residuals from the smooth, smoothing these and adding the two smoothed series together.)

VALUE:

smoothed vector using 4(3RSR)2H twice.

REFERENCE:

Chapters 7 and 16 of *Exploratory Data Analysis*, J. W. Tukey, Addison Wesley.

solve: Solve Systems of Linear Equations

`solve(a, b)`

ARGUMENTS:

- a:** matrix of coefficients. Must be square and non-singular.
b: optional matrix of coefficients. If *b* is missing, the inverse of matrix *a* is returned.

VALUE:

the solution *x* to the system of equations $a \%* x = b$.

EXAMPLES:

`ainv ← solve(a) #invert a`

sort: Sort into Ascending Order

`sort(data)`

ARGUMENTS:

data: vector.

VALUE:

vector with its data sorted in ascending order. Character data is sorted in lexicographic order.

The result is a vector, even if *data* was a structure (matrix, time-series).

The ordering of character data depends on the character code. For ASCII code, digits precede upper-case letters, which precede lower-case letters. The position of other characters is unintuitive.

source: Input from a File

`source(file)`

ARGUMENTS:

file: character string giving the name of a file. Commands are read from the file, instead of from the terminal. This continues until an end-of-file is encountered, until any error (syntax or execution) is generated, or until *source* with a null argument is encountered. The argument *sourcexit* to function *options* can be used to allow execution to continue after errors in a source file.

Source files can themselves invoke source files, macros, etc.. This facility is recursive.

Command input is taken from the source file when the parser reads the next line. Therefore, the line

`source("abc"); 1+2`

will evaluate $1+2$ before reading anything from file "abc".

split: Split Data by Groups

```
split(data, group)
```

ARGUMENTS:

data: vector structure containing data values to be grouped. Missing values (NAs) are allowed.

group: vector or category structure giving the group for each data value. For example, if the third value of *group* is 12, the third value in *data* will be placed in a group with all other data values whose group is 12.

VALUE:

structure in which each component contains all data values associated with a group. Within each group, data values are ordered as they originally appeared in *data*. The name of the component is the corresponding value in *group*, or the corresponding category name.

EXAMPLES:

```
split(people, age%10) # note integer divide
split(gnp, cycle(gnp)) # component for each month
split(student, grade)
boxplot(split(income, month))
```

sqrt: (see *exp*)

rstab: Stable Family of Distributions

```
rstab(n, par1, par2)
```

ARGUMENTS:

n: number of random values desired.

par1: parameter for the member of the stable family desired. These are specified in the form given in the reference. The parameter *par1* is usually called the index of the family: 2 corresponds to the normal, 1 to the Cauchy. Generally, smaller values mean longer tails. Only values between 0 and 2 are legal.

par2: modified skewness parameter (see the reference). Negative and positive values correspond to skewness left and right.

VALUE:

vector of *n* values from the chosen family.

Stable distributions are of considerable mathematical interest. Statistically, they are used mostly when an example of a very long-tailed distribution is required. For small values of *par1*, the distribution degenerates. See the reference and other works cited there.

Note that there are no probability or quantile functions for this distribution. The efficient computation of such values is an open problem.

REFERENCE:

Chambers, Mallows, and Stuck, *JASA*, 1976, pp 340-344.

EXAMPLES:

```
hist(rstab(200, 1.5, 1.5)) # fairly long tails, skewed right
```

stare: (see photo)

stars: Star Plots of Multivariate Data

stars(x, full, scale, radius, type, labels, head, max, byrow)

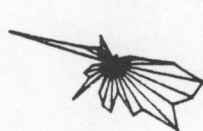
ARGUMENTS:

- x:** matrix of data. One star symbol will be produced for each row of the matrix.
- full:** logical, if TRUE, the symbols will occupy a full circle. Otherwise, they occupy the (upper) semi-circle only. Default TRUE.
- scale:** logical, if TRUE, the columns of the data matrix are scaled so that the maximum value in each column is 1 and the minimum 0. If FALSE, the presumption is that the data has been scaled by some other algorithm to the range $0 \leq x[i,j] \leq 1$. Default TRUE.
- radius:** logical, if TRUE (default), the radii corresponding to each variable in the data will be drawn (out to the point corresponding to $x[i,j] = 1$).
- type:** optional character string, giving the type of star to draw. Reasonable values are "l", "p", "b" for lines, points and both. Default is "l".
- labels:** optional character vector for labelling the plots. By default, labels are "1", "2", etc. If supplied, the label vector must have length equal to `nrow(x)`.
- head:** optional character string for a running head. This is plotted as a title at the top of each page. By default, the name of the data is used. If there is more than one page, the page number is included in the title.
- max:** optional, suggested limit for the number of rows or columns of plots on a single page. Default forces all symbols to be on one page. The algorithm tries to choose an array of plots which efficiently uses the available space on the display.
- byrow:** logical flag, should the symbols be plotted row-by-row across the page, or column-by-column? Default FALSE.

EXAMPLES:

```
# the example plot is produced by:
stars(votes.repub[state.region == 1,], radius=T,
labels=encode(state.name[state.region == 1]),
head="Republican Votes (Northeast) 1856 - 1976")
```


Republican Votes (Northeast) 1856 - 1978



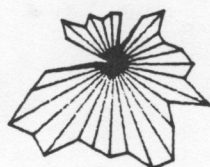
Connecticut



New Hampshire



Pennsylvania



Maine



New Jersey



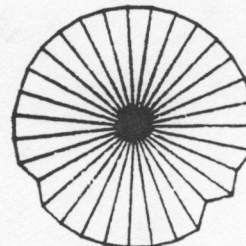
Rhode Island



Massachusetts



New York



Vermont

starsymb: Plot a Single Star Symbol

```
starsymb(x, full, scale, radius, type, collab, sample)
```

ARGUMENTS:

x: data matrix, as passed to stars.
full: logical, TRUE (the default) if full 360 degree symbols wanted.
scale: logical, TRUE (the default), if the columns of the matrix should be scaled independently.
radius: logical, if TRUE (default), radii corresponding to each variable will be drawn.
type: the type of the plotted symbol (see *stars*).
collab: vector of character string labels for the variables (columns of *x*).
sample: which of the star symbols for the data *x* (row of *x*) should be shown to illustrate the symbol?
 Default 1.

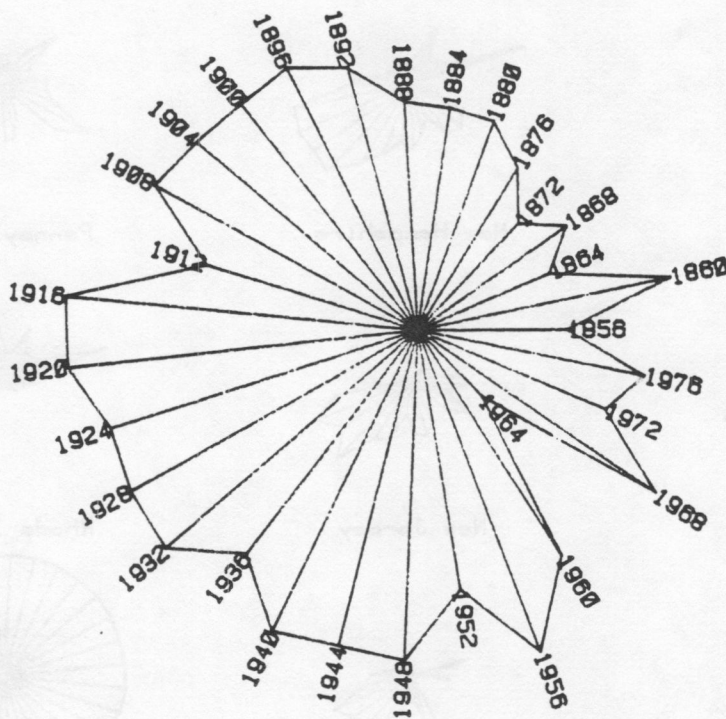
EXAMPLES:

```

starsymb(votes.repub,collab=encode(votes.year),sample=30,scale=F)
#plot the republican votes from New Jersey
title(main=encode("Republican Votes for",state.name[30]))

```

Republican Votes for New Jersey



start: (see end)

stem: Stem and Leaf Display

stem(x, nl, scale, twodig, fence, head)

ARGUMENTS:

- x: numeric vector to be displayed. Missing values (NAs) are allowed.
- nl: number of different leaf values on a stem. Allowed values are 2, 5, 10. Default is to determine an appropriate value automatically.
- scale: position at which break occurs between stem and leaf, counting to the right from the decimal point; e.g., -1 would break between the tens and the units digit. By default, a suitable position is chosen from the range of the data.
- twodig: number of leaf digits, 1 or 2 (default 1).
- fence: the multiple of the inter-quartile range used to determine outliers. By default, any point further than 2 inter-quartile ranges from the nearest quartile is considered an outlier, and is printed separately from the body of the stem and leaf display.
- head: logical, default TRUE, which controls heading giving median, hinges, and counts of data values and NAs.

EXAMPLES:

```
stem(iris, nl = 5)
```



```
# the example plot is produced by:
my.sample←rt(50,5)
stem(my.sample)
```

N = 50 Median = -0.0970237 Hinges = -0.776103, 0.5637632

Decimal point is at the colon

```
 2      2    -1 : 95
 8      6    -1 : 443211
18     10    -0 : 9888877765
      10    -0 : 4443321110
22      8     0 : 00133344
14      6     0 : 568999
 8      4     1 : 0134
 4      2     1 : 99
 2      0     2 :
 2      1     2 : 9
```

High: 3.603643

S Data Structures

S provides both general hierarchical data structures and automatic handling of commonly occurring statistical data. Data structures are of three degrees of complexity.

ARGUMENTS:

Vector: may be of mode LGL (logical), INT, REAL or CHAR. The last may not be used for numerical data. Each element of a CHAR vector is a string.

Vector-structure: may be either a vector or a structure with one of its entries a vector named *Data*. The two most important examples are TS (time-series) and ARRAY, including matrices.

Numeric-structure: vector structure, except one for which the *Data* entry is of mode CHAR.

General: consists of an arbitrary set of components, each of which may be a vector, a vector structure, or a general structure. The components have names, so that one may refer to them by their component names (see *select*).

EXAMPLES:

```
i ← 1:10; x ← i**.5; t ← "abc"
j ← cstr(Data=1:10,t="abc")
k ← cstr(first=j,second=x); m ← cstr(first=i,second=x)
(i, x and t are vectors; j is numeric; k and m are general,
but not numeric: k has a Data, but 2 levels down; m has no Data.)
```

[]: Subsets of Data

`data [expr1, expr2, ...]`

ARGUMENTS:

data: a matrix, array or vector structure.

expr1: either a logical vector of same length as *data* or an (integer) vector of subscripts, in which case the subscripts must be all positive or all negative. Positive values select those elements; negative values select all except those elements.

VALUE:

a vector of same mode as *data* containing either the elements for which *expr1* is TRUE or the elements given by the integer *expr1*.

EXAMPLES:

```
x[x!=999.999]
x[order(y)]
x[-c(1,3)] #all but the first and third
```

In the case of arrays, the first expression selects on the first dimension, the second on the second, etc. The number of expressions must be either the number of dimensions in the array, or else one. In the second case, the array is subscripted as a vector.

EXAMPLES:

```
If A is an array with dimension (5,3,2)
A[1,1,1] is a scalar, the first data value of A
A[1] is the same
A[,1:2,] is a (5,2,2) array
A[A>3] is a vector
```

subtree: Extract Part of a Cluster Tree

`subtree(tree, leaves)`

ARGUMENTS:

tree: a cluster tree structure, normally the result of a call to *hclust*.

leaves: vector of objects (i.e., row numbers of the original data matrix) which should be included in the extracted subtree.

VALUE:

the smallest subtree of *tree* which includes all the *leaves*. The subtree includes the components *merge*, *height* and *order* described under *hclust*. It can be used in the various summaries of cluster results, such as *plclust*.

EXAMPLES:

```
z ← hclust(dismat)
subtree(z,c(1,10)) #subtree including 1st and 10th objects
```


sum: (see **prod**)

svd: Singular Value Decomposition

```
svd(y, nu, nv)
```

ARGUMENTS:

y: matrix of arbitrary size.
nu: optional number of columns wanted in matrix *u*, may be zero. Default is `min(nrow(y), ncol(y))`.
nv: optional number of columns wanted in matrix *v*, may be zero. Default is `min(nrow(y), ncol(y))`.

VALUE:

structure containing the components of the singular value decomposition, $y = u \%* d \%* t(v)$
u: if $nu > 0$, an `nrow(y)` by `nu` matrix of unit orthogonal columns. Missing if $nu = 0$.
d: the vector of singular values (diagonal elements of matrix *d*).
v: if $nv > 0$, `ncol(y)` by `nv` matrix of unit orthogonal columns. Missing if $nv = 0$.

sweep: Sweep Out Array Summaries

```
sweep(a, margin, stats, fun, arg ...)
```

ARGUMENTS:

a: array. Missing values (NAs) are allowed.
margin: vector describing the dimensions of *a* that correspond to *stats*.
stats: vector giving a summary statistic of array *a* that is to be swept out. Missing values (NAs) are allowed.
fun: character string name of function to be used in sweep operation. Default "-" (subtraction).
arg: optional arguments to *fun*.

VALUE:

an array like *a*, but with marginal statistics swept out as defined by the other arguments. In the most common cases, the function is "-" or "/" to subtract or divide by statistics that result from using the *apply* function on the array. For example: `colmean—apply(z,2,"mean")` computes column means of array *z*. `zcenter—sweep(z,2,colmean)` removes the column means.

More generally, based on *margin*, there are one or more values of *a* that would be used by *apply* to create *stats*. *sweep* creates an array like *a* where the corresponding value of *stats* is used in place of each value of *a* that would have been used to create *stats*. The function *fun* is then used to operate element-by-element on each value in *a* and in the constructed array.

EXAMPLES:

```
a ← sweep(a,2,apply(a,2,"mean")) # remove col means
a ← sweep(a,1,apply(a,1,"mean")) # row means
# a simple two-way analysis
```

symbols: Draw Symbols on a Plot

`symbols(x, y, circles=, squares=, rectangles=, stars=, add, inches)`

ARGUMENTS:

`x,y`: coordinates of centers of symbols. A structure containing components named `x` and `y` can also be given.

`circles=`: the radii of the circles.

`squares=`: the lengths of the sides of the squares.

`rectangles=`: matrix whose two columns give widths and heights of rectangles.

`stars=`: a matrix with n columns, where n is the number of points to a star. The matrix should be scaled from 0 to 1.

`add`: logical flag. If FALSE (default), a new plot is set up. If TRUE, symbols are added to the existing plot.

`inches`: if FALSE, the symbol parameters are interpreted as being in units of `x` and `y`. If TRUE (default), the symbols are scaled so that the largest symbol is one inch in size. If a number is given, the parameters are scaled so that *inches* is the size of the largest symbol in inches.

Graphical parameters may also be given as arguments to this function (see *par*).

Only one of the arguments `circles=`, `squares=`, `rectangles=`, or `stars` can be given.

EXAMPLES:

```
symbols(x,y,circle=radii,inches=5) #largest circle has radius 5 inches
```

```
p ← ?col(parm,range) #find min/max of each column of parameter matrix
```

```
pscale ← scale(p,center=p[1,],scale=p[2,]-p[1,]) # scale to 0-1
```

```
symbols(x,y,stars=pscale) # plot stars
```

S Syntax: All Commands to S are Expressions

The simplest expressions are

number 1, 5.3, 1e17

character string "abc", 'def ghi'

name iris, city23

More complicated expressions can be made by combining expressions using the following rules:

expr op expr where *op* is + - * / ^ ** == != <= >=
 < > & : %%

(*expr*) parens specify order of evaluation

op expr unary operators + - !

name ← *expr* assignment

expr \$ *name* selection of component

name [*expr*, *expr*, ...] subscripting

name (*name1*=*expr1*, *name2*=*expr2*, ...) function call

name \$ [*expr*] component by number

for(*name* in *expr*) *expr* loop

if (*expr*) *expr1* else *expr2* conditional expression

{ *expr1*; *expr2*; ... } compound expression

Notice that expressions can be arbitrarily complicated by repeated uses of the above rules.

A typed expression may be continued on further lines by ending a line at a place where the line is obviously incomplete with a trailing comma, operator, or with more left parens than right parens (implying more rights will follow). Ordinarily the prompt character signifying that more input is necessary is "> ", but when continuation is expected the prompt is "+ ".

The *name* for subscripting may also be a parenthesized *expr*, function call, or component selection.

Functions may be called without arguments, may have missing arguments of the form *name*= or ,, and the keywords *name*= are optional.

The value of the input expression is automatically printed if it is not saved, producing a "desk calculator" mode. The expression *A*←1+2 is not printed because the result is saved in *A*. Certain functions (*save*) do not automatically print their results.

A line consisting of only a name is first interpreted as a function with no arguments, and if the function is not found, the line is treated as a dataset name and automatically printed.

If the top-most *expr* is a function call, it is not necessary to use parens to delimit its argument list, e.g. *plot* *x,y* is identical to *plot*(*x,y*). All other functions must use parens, however.

A line whose first character is "!" is executed as a system command with no changes.

The value of a compound expression is the value of the last executed subexpression.

sys: Execute System Commands

sys(cmd)

ARGUMENTS:

cmd: character vector, each element of which is passed to the operating system and executed as a system command.

EXAMPLES:

sys("cat myfile") # same as !cat myfile

S System File and Source Code Organization

S organizes its system files for source code, libraries, utilities and functions by exploiting the hierarchical file structure in the operating system. The various directories to be described each contain files related to specialized parts of S. In each directory source code for interface routines and algorithms will be under file names ending in ".i", ".r", etc. as described in sections 6 and 7. Files (whether source or other) which are dependent on the UNIX operating system will be in a subdirectory "u" of the directory.

On any installation, all the S files will be under the S home directory. The name of this directory is installation dependent (see Note below): for this documentation we will refer to it as SHOME. At the top level, the S home directory is divided into subdirectories: Include, cmd, grz, guide, lib, psl, s, and tmp.

- Include: All the macro files accessible in the interface and algorithm languages. This is the directory accessed by the INCLUDE statement.
- cmd: The executable commands (utilities) in S. The command "S name" executes file "name" in this directory.
- grz: The source code for all graphical algorithms supplied with S. This directory is further subdivided into a number of directories. The interesting ones are: *grzsource* for the general graphical algorithms; *standalone* for source for stand alone graphics; and various directories for device driver code, of the form "*devsource*", where *dev* is the device name (e.g., *printersource* for printer device code).
- guide: Files containing text for this manual. Subdirectories *basic*, etc. contain text for corresponding sections of the manual. Subdirectory *man* contains the manual page documentation for S.
- lib: The object libraries for the S system algorithms.
- psl: Source code for *portable statistical library* of routines in the algorithm language. These routines can be used outside of S (see documentation for PROGRAM).
- s: All the files directly related to the S system itself. The relevant subdirectories are as follows.
 - s/.help: All the detailed documentation files.
 - s/basic: Source code for all S functions except macro and graphics.
 - s/data: The system shared data base.
 - s/graph: Source code for all S graphical functions.
 - s/macro: Source code for the S macro processor.
 - s/main: Source code for the S executive.
 - s/source: Source code for the algorithms supporting the S system itself.
 - s/util: Source code for utilities, e.g. *extract* (shell files are under *cmd* above).
 - s/x: The executable version of all S functions.
- tmp: The temporary expansions of interface and algorithm routines produced by MAKE. Look here for the lines on which ratfor, f77 and c report errors.

The S home directory name differs depending on the local installation. The following trick will print the directory name. Type

S.

which will try to execute an unexecutable file in *cmd*. The error message might, for example, be:

S: cannot execute /usr/src/s/cmd/.

This tells you that "/usr/src/s" is the S home directory.

t: Transpose a Matrix

`t(x)`

ARGUMENTS:

x: matrix. Missing values (NAs) are allowed.

VALUE:

transpose of *x* (rows of *x* are columns of result).

table: Create Contingency Table from Categories

`table(arg1, arg2, ...)`

ARGUMENTS:

arg: categories, as variables for the contingency table.

VALUE:

contingency table structure, i.e., a multi-way array with an additional component *Label*. This is a structure with one component for each *arg*; the component corresponding to *arg1* is named "*arg1*" and is a character vector containing the values from *arg1*\$*Levels*.

EXAMPLES:

```
table(age,sex,race,income) # all variables previously processed
# by functions code or cut
```

tbl: Write a table to be processed by the "tbl" system

`tbl(x, file, head, rowlab, collab, options)`

ARGUMENTS:

x: matrix to be printed as a table. May be any mode and may contain NA's.
file: name of the file on which the *tbl* output will be printed. Default is "tbl.out".
head: optional character string to appear as the heading of the table.
rowlab: optional character vector to be used as labels for the rows of the table.
collab: optional character vector to be used as labels for the columns of the table.
options: optional vector of options to the "tbl" system (see the operating system manual). Default is "center,box;".

The output file must be used with the "tbl" preprocessor to *nroff*/*troff*. The *tbl* function in S is currently dumb about things like long lines or long fields. Either edit the output file or (better) break the matrix up into reasonable subsets of columns.

EXAMPLES:

```
tbl(x) #simple table
tbl(x>0,r=encode("Row",seq(10)),h="Is x positive?")
```

pt qt rt: t-Probability Distribution

pt(q, par1)
qt(p, par1)
rt(n, par1)

ARGUMENTS:

q: vector of quantiles.
p: vector of probabilities.
par1: vector of degrees of freedom.
n: sample size.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*rt*), from Students t-distribution on *par1* degrees of freedom.

tek10: (see **tek12**)

tek10 tek12 tek14 tek14q: Tektronix Storage Scope Devices

tek10(ask)
tek12(ask)
tek14(ask)
tek14q(ask)

ARGUMENTS:

ask: logical, should device driver print the message "GO?" to ask permission to clear the screen?
Default TRUE.

tek10, *tek12*, *tek14*, and *tek14q* identify Tektronix storage scope graphics devices. *tek10* is designed for upper-case only models (4006, 4010), *tek12* for upper/lower case models such as 4012, while *tek14* and *tek14q* are designed specifically to support 4014 models. *tek14q* is somewhat quicker in its plotting than *tek14*, at the cost of slightly decreased resolution.

If *ask* is TRUE, whenever a new plot is about to be produced the message "GO?" appears in the lower left corner of the screen. This allows further viewing, making a hard copy, etc., before the screen is erased. When ready for plotting to proceed, simply hit carriage return. The function *hardcopy* can be used to automatically generate a copy of the information on the screen. The *hardcopy* function can be used with *ask=FALSE* to produce unattended plots.

When graphic input (*identify*, *rdpen*) is requested from these terminals, the cursor lines appear horizontally and vertically across the entire screen. The thumb wheels should be adjusted to position the intersection of the cursors at the desired point, and then any single character should be typed (may need carriage return on some terminals). A carriage return alone will terminate graphic input without transmitting the point.

Only *tek14* and *tek14q* change character size and line type. Character sizes are .9, 1, 1.5, and 1.6. There are 5 line types (1 through 5) giving solid, dotted, dot dash, short dash, and long dash lines. Characters can not be rotated on these devices.

A device must be specified before any graphics functions can be used.

tek14: (see tek12)

tek14q: (see tek12)

tek46 tek46h tek46v: Tektronix 4662 Pen Plotter

tek46(width, height, ask, color)
tek46h
tek46v

ARGUMENTS:

width: width of plotted surface in inches. Default is 8 for *tek46v*, 10 for *tek46h*, 15 otherwise.

height: height of plotted surface in inches. Default is 8 for *tek46h*, 10 otherwise.

ask: logical, should user be prompted by "GO?" prior to advancing to new frame? Default TRUE.

color: logical, should user be prompted to change pen colors whenever the graphical parameter *col* changes? Default FALSE.

These commands identify Tektronix 4662 series plotters. The plotter may be used with standard 8.5 by 11 inch paper with the long dimension horizontally (*tek46h*) or vertically (*tek46v*), or with 11 by 17 paper (*tek46*). The arguments listed after *tek46* can also be given with *tek46h* or *tek46v*.

Whenever a new plot is about to be produced, the message "GO?" appears on the terminal. At this time, new paper may be loaded, pens changed, etc. When ready for plotting to proceed, simply hit carriage return.

When graphic input (*identify*, *rdpen*) is requested, the plotter will beep and light its "prompt" light. The pen should be positioned by means of the joystick. When the desired pen position is reached, depress the "call" button momentarily. Hold the "call" button down for about 2 seconds (until it beeps once) to terminate the input. The coordinates of the terminating point are not transmitted.

Character sizes may be changed to any desired value. Pens can be changed manually to provide different colors. If argument *color* is TRUE, the user is prompted with the message "Load pen x" when graphical parameter *col* changes to x. The initial pen is assumed to be color 1. Characters can be rotated to any orientation. Only solid line style is available.

At 300 baud, the switch settings on the back of the device should be set to "0231" (no parity) or "02B1" (even parity).

A device must be specified before any graphics functions can be used.

text: Plot Text

```
text(x, y, label)
```

ARGUMENTS:

- x,y:** vector of position(s) at which to plot labels. A time series or structure containing *x* and *y* may also be given for *x*. Missing values (NAs) are allowed.
- label:** controls labelling. The interpretation depends on the mode of *label*. If it is a character vector, *label[i]* is plotted at each point (*x[i]*,*y[i]*). If it is a logical vector (of the same length as *x* and *y*), the value *i* is plotted at each (*x[i]*,*y[i]*) for which *label[i]* is TRUE. If *label* is integer or real, *label[i]* is encoded and plotted at (*x[i]*,*y[i]*). Missing values (NAs) are allowed.

If the lengths of *x*, *y*, and *label* are not identical, the shorter vectors are reused cyclically.

The *text* function can be used after *plot(x,y,type="n")* which draws axes and surrounding box without plotting the data.

EXAMPLES:

```
plot(x,y,type="n")
text(x,y,seq(x)) #plot i at (x[i],y[i])
```

time cycle: Create Time Vector

```
time(x)
cycle(x)
```

ARGUMENTS:

- x:** time series. Missing values (NAs) are allowed.

VALUE:

time series with same *start*, *end* and *nper* as *x*. *time* returns the time value at each point, e.g., the value at January, 1973 is 1973.0, and the value at July 1973 is 1973.5. *cycle* returns the period associated with each observation, e.g., 1 for each January, etc.

EXAMPLES:

```
split(gnp,cycle(gnp)) #structure with component for each month
```

title axes: Plot Titling Information and/or Axes

```
title(main, sub, xlab, ylab, axes)
axes(main, sub, xlab, ylab, axes)
```

ARGUMENTS:

- main:** optional character string for the main title, plotted on top in enlarged (*cex*=1.5) characters.
- sub:** optional character string for the sub-title, plotted on the bottom.
- xlab:** optional character string to label the x-axis.
- ylab:** optional character string to label the y-axis.
- axes:** logical, should axes be drawn? Default TRUE for function *axes*, FALSE for *title*.

Graphical parameters may also be supplied as arguments to this function (see *par*).

The two functions differ only in that *axes* also plots the *x* and *y* axes. All the character strings are empty by default.

EXAMPLES:

```
title("residuals from final fit") # main title only
axes(xlab="concentration",ylab="temperature")
```

tprint: Print Multi-way Tables

```
tprint(table1, table2, ..., Label=, maxcol=, mincol=)
```

ARGUMENTS:

- table:** table or multi-way array. Contingency tables may be constructed (complete with *Label* component) from category data by the *table* function. If more than one table is given, values from the tables are printed above one-another, and the values are labelled by the table name, or by the *name* used if *table* is given in *name=value* form. All tables must have the same dimensionality and number of levels on each category (i.e., dimension of the tables).
- Label=:** optional structure containing as many components as there are categories to the tables. The name of each component is the name used for the corresponding category, and each component should be a character vector with as many values as the number of levels of the category. If any of the tables already contain such a component named *Label*, this argument is not needed.
- maxcol=:** optional, maximum category to be laid out across the page. *tprint* displays the first category of the tables down the page and the second across the page. If there is enough room, categories 3, 4, ... will also be laid out across the page. If *maxcol* is given this will be the last category laid out this way, even if more would fit.
- mincol=:** optional, minimum category to be laid out across the page. Even if the category will not fit, *tprint* will lay it out across the page, breaking the table into blocks if necessary. By default, no category will be used across the page unless all levels fit.

EXAMPLES:

```
mydata ← table(age,sex,height,weight,eye.color)
tprint(mydata) # print the table with labels for the levels
fitted ← loglin(mydata,model) # fit log-linear model to table
tprint(data=mydata,fit=fitted,residuals=mydata-fitted)
# three tables: data, fit, residuals for each cell
# the example plot is produced by:
tprint(table(age,sex,weight,height))
```

height: Short	<= 50		50 to 100		100 to 150		> 150	
weight:								
sex:	Male	Female	Male	Female	Male	Female	Male	Female
age:								
<= 10	2	1	1	0	1	0	1	2
10 to 20	1	0	1	0	2	2	1	1
20 to 30	1	0	2	0	0	2	1	0
30 to 40	1	0	2	1	1	0	1	1
40 to 50	0	1	3	0	1	2	0	1
50 to 60	0	0	0	0	2	0	0	1
60 to 70	0	0	0	0	0	0	0	0
> 70	0	0	0	0	0	0	0	0

height: Medium	<= 50		50 to 100		100 to 150		> 150	
weight:								
sex:	Male	Female	Male	Female	Male	Female	Male	Female
age:								
<= 10	0	1	0	1	0	0	2	2
10 to 20	3	0	0	1	1	1	0	0
20 to 30	0	2	0	1	2	2	0	0
30 to 40	0	0	0	0	0	1	1	0
40 to 50	0	0	0	2	0	0	0	0
50 to 60	2	1	0	0	1	0	0	0
60 to 70	0	0	0	0	0	0	0	0
> 70	0	0	0	0	0	0	0	0

height: Tall	<= 50		50 to 100		100 to 150		> 150	
weight:								
sex:	Male	Female	Male	Female	Male	Female	Male	Female
age:								
<= 10	1	0	0	2	0	1	3	1
10 to 20	0	1	2	2	0	2	1	0
20 to 30	1	0	1	0	1	0	0	0
30 to 40	2	0	3	1	0	0	1	0
40 to 50	0	0	0	0	0	1	0	1
50 to 60	0	1	0	0	1	2	1	0
60 to 70	0	0	0	0	0	0	0	0
> 70	0	0	0	0	0	0	0	0

trunc: (see ceiling)

ts: Create Time Series

ts(data, start, nper, end, tsp)

ARGUMENTS:

- data:** numeric vector giving the data values for the time series.
- start:** starting date for the series, in years, e.g., February, 1970 would be 1970+(1/12) or 1970.083. If *start* is a vector with at least two data values, the first is interpreted as the year, and the second as the number of *nper*s into the year; e.g., February, 1970 could be c(1970,2).
- nper:** periodicity of the series, in observations per year, e.g. monthly data has *nper*=12, yearly has *nper*=1.
- end:** ending date for the series.
- tsp:** real vector giving the values for *start*, *end* and *nper*. *tsp* may be used in place of these 3 arguments.

VALUE:

time series with the given *data* and *start*, *end*, *nper*.

ts checks that the number of observations is compatible with the specified dates. Default values for *start*, *end* and *nper* are 1, *len(data)*, and 1. If only two of *start*, *end*, and *nper* are given, the

other parameter will be computed based on the given values and `len(data)`.

Any periodic data can be fit into the time series framework. Hourly data might use days for `start` and `end` with `nper=24`; daily data could use weeks with `nper=7`.

tslines: (see `tsplot`)

tsmatrix: Create Matrix with Time Series as Columns

```
tsmatrix(arg1, arg2, ...)
```

ARGUMENTS:

argi: time series to form a column of the resulting matrix. The time window for the matrix will be the intersection of time windows for the arguments (i.e., the maximum of the start dates and the minimum of the end dates). All series must have the same periodicity.

VALUE:

a matrix with one column for each argument and a component named *Tsp* with the start date, end date, and number of observations per year.

Note the distinction from the related function *cbind*. This allows vectors or matrices as arguments as well, but makes no effort to make time parameters consistent. Frequently, the two functions can be used together (see examples).

EXAMPLES:

```
tsmatrix(x,lag(x),diff(x)) # x, lag-1 of x and first diffs
cbind(tsmatrix(x,lag(x)),1) # two series, column of 1s
```

```
yx<-tsmatrix(employment,gnp,lag(gnp), ...)
regress(yx[,1],yx[,-1]) # regression with x and y times aligned
```

tsplot tslines tspoints: Plot Multiple Time Series

```
tsplot(ts1, ts2, ..., type=, lty=, pch=, col=)
```

ARGUMENTS:

tsi: time series to be plotted. The x-axis is set up as the union of the times spanned by the series. The y-axis includes the range of all the series. Missing values (NAs) are allowed.

type=: an optional character string, telling which type of plot (points, lines, both, none or high-density) should be done for each plot. The first character of type defines the first plot, the second character the second, etc. Elements of type are used cyclically; e.g., "pl" alternately plots points and lines. Default is "l" (lines) for *tsplot* and *tslines* and "p" for *tspoints*.

lty=: optional vector of line types. The first element is the line type for the first line, etc. Line types will be used cyclically until all plots are drawn. Default is 1,2,3,4,5.

pch=: optional character vector for plotting characters. The first character is the plotting character for the first plot of type "p", the second for the second, etc. Default is the digits (1 through 9, 0) then the letters.

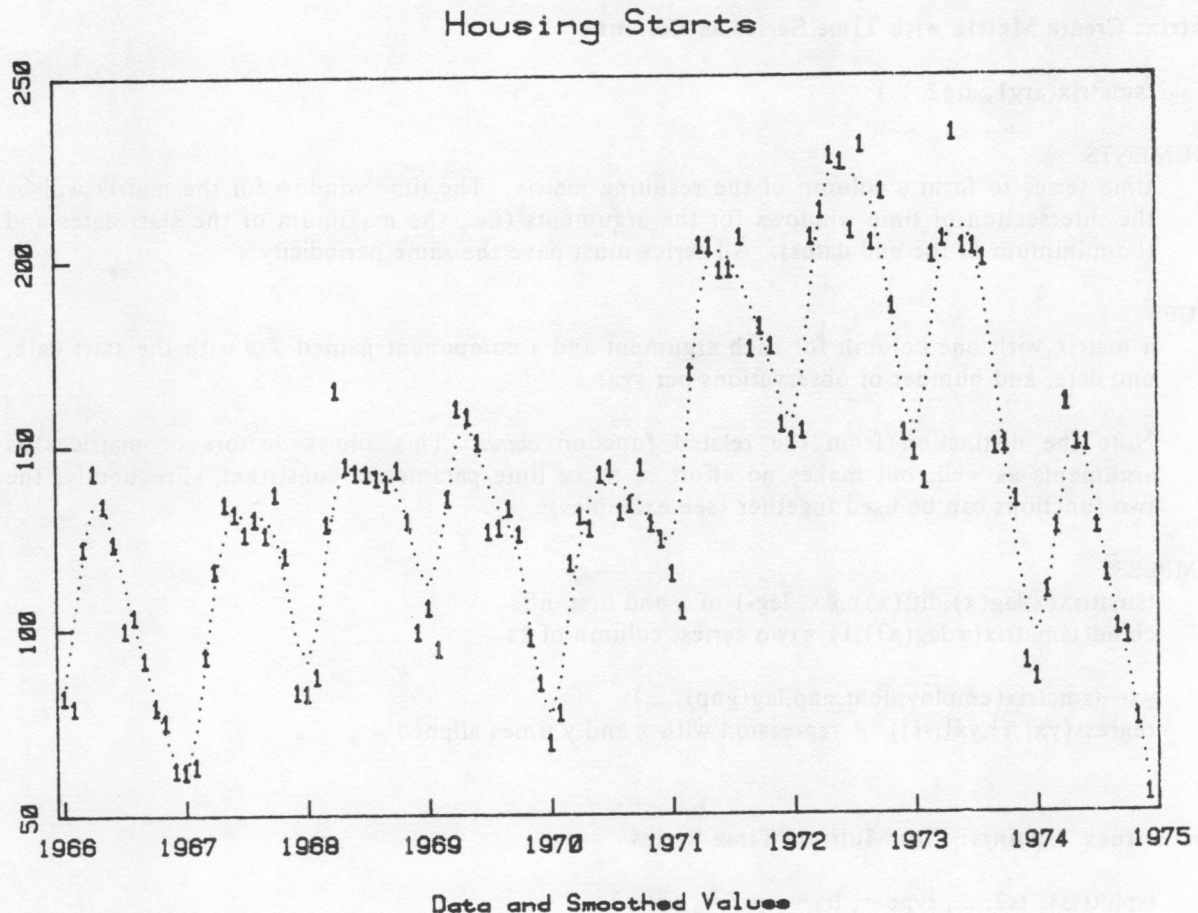
col=: optional vector of colors. Default is 1,2,3,4. Colors are also cycled if necessary.

Graphical parameters may also be supplied as arguments to this function (see *par*).

tsplot generates a new plot; *tspoints* and *tslines* add to the current plot.

EXAMPLES:

```
tsplot(gnp,smooth(gnp),type="pl")
tspoints(x,y,pch="*") # points with "*" for both series
# the example plot is produced by:
tsplot(hstart,smooth(hstart),type="pl")
title(main="Housing Starts",sub="Data and Smoothed Values")
```



tspoints: (see *tsplot*)

twoway: Analysis of Two-way Table

```
twoway(x, trim, iter, eps, print)
```

ARGUMENTS:

- x:** matrix to be analyzed. Missing values (NAs) are allowed.
- trim:** optional trimming fraction for carrying out analysis. Default .5 trimmed mean (median). *trim=0* will cause analysis by means, .25 by midmeans, etc.
- iter:** maximum number of full (row and column) sweeps. Default 6.
- eps:** error tolerance. If *eps* is given, the algorithm will iterate until the maximum change in row or column effects is $< \text{eps}$. Default is to iterate until the specified number of iterations or until converged to the accuracy of the machine arithmetic. It is not always possible to converge to a unique answer.
- print:** logical flag, should convergence printing be done? Default, FALSE. If TRUE, the maximum change in row/column effects in the last iteration is printed.

VALUE:

structure consisting of 4 components, *resid*, *row*, *col*, and *grand*, such that

$$x[i,j] = grand + row[i] + col[j] + resid[i,j]$$

grand: overall location estimate of the data.

row: vector of row effects.

col: vector of column effects.

resid: matrix of residuals from the fit.

The function *plotfit* produces a graphical display of the fit generated by *twoway*.

EXAMPLES:

```
twoway(temperature,trim=.25) # analysis by midmeans
```

punif qunif runif: Uniform Distribution.

```
punif(q, par1, par2 )
```

```
qunif(p, par2, par2 )
```

```
runif(n, par1, par2 )
```

ARGUMENTS:

q: vector of quantiles.

p: vector of probabilities.

par1: vector of lower limits. Default is 0.

par2: vector of upper limits. Default is 1.

VALUE:

probability (quantile) vector corresponding to the given quantile (probability) vector, or random sample (*runif*), for the uniform distribution on the range *par1* to *par2*.

EXAMPLES:

```
runif(100,-1,1) #100 numbers uniform on -1 to 1
```

uniq: (see **unique**)

unique uniq: Unique Values in a Vector

```
unique(x)
```

```
uniq(x)
```

ARGUMENTS:

x: vector

VALUE:

a vector like *x* but with no duplicate values. The values will be in the same order as *x* except that repeated values will be deleted.

EXAMPLES:

```
sort(unique(names)) #sorted list of names with no duplicates
```

usa: Plot United States Coast-line and State Boundaries

```
usa(states, coast, add, xlim, ylim)
```

ARGUMENTS:

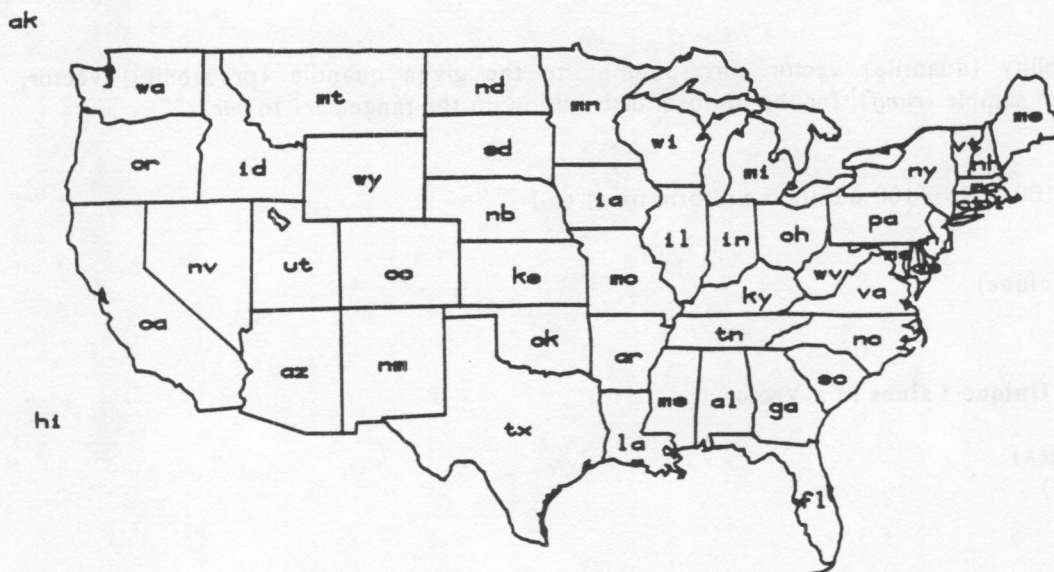
states: logical flag to control whether state boundaries are plotted. Default TRUE.
coast: logical flag to control whether coast-line is plotted. Default TRUE.
add: logical flag. If TRUE, plot is superimposed on existing plot. Otherwise, a new plot is generated. Default FALSE.
xlim: optional limits for the x-axis (longitude). Default is c(65,135).
ylim: optional limits for the y-axis (latitude). Default is c(24,50).

Graphical parameters may also be supplied as arguments to this function (see *par*).

The plot is done in correct physical proportion. The coordinate system set up for the plot uses negative longitude, so that x-values increase from left to right on the plot.

EXAMPLES:

```
usa(states=F) # the u.s. without state lines  
usa(xlim=c(65,85),ylim=c(35,50)) #plot the north-east  
# the example plot is produced by:  
usa  
text(state.center,state.abb)
```



var: (see **cor**)

vu: Create Vu-graphs (Overhead Slides) on a Plotter

`vu(text, indent, width, height, line)`

ARGUMENTS:

text: character string vector, the elements of which are either lines of text to plot, or else commands to the *vu* function (see below). The text will be plotted, with character size as large as possible to fill the page. The best way to initialize a vugraph dataset is usually with the *edit* function; for example

`vutest ← ".C1"; edit(vutest)`

indent: number of spaces to indent lists. Default 3.

width: width of the page in inches. Default 7.

height: height of the paper in inches. Default 7.

line: thickness of the pen line in inches (needed for bold face and for drawing bullets - see below). Default is .015 inches (the thickness of the thick pens for the HP7221). Thin pens on the HP7221 are .01 inches.

The following are the commands recognized by *vu*. They are similar to commands in several *troff* macro systems, and they begin with a period (.) and one or two capital letters:

- .C** Choose a color. This command takes an argument which is the color number, i.e., ".C 3" for color 3, etc.
- .L** Make the text larger. Each time this command appears, the size is increased by 1/4 the initial size of the text (so that two commands increase the size by 1/2, four double it, etc.) This command may also take an argument, which is the desired character size relative to initial size 1, thus ".L 2" would be double the initial size, and ".L 1" would return to initial size.
- .S** Make the text size smaller, by the same amount that ".L" makes it larger. Thus, any number of ".S" commands cancel the same number of ".L" commands. This command may take an argument, in which case it behaves exactly as ".L".
- .B** Embolden the text (done by replotting one line width to the right). The argument tells how many overstrikes to use, default 1.
- .R** Remove the emboldening.
- .CE** The argument gives the number of following lines of text which are to be centered, default 1.

The next commands all create lists of items. The items in the list can be preceded by bullets, diamonds, numbers, or any character string (for example, "--"). Lists may be nested, but general or numbered lists may only appear once.

- .BL** Start a bullet list.
- .DL** Start a diamond list.
- .GL** Start a general list with each item preceded by the character string given after the command.

For long strings, the value of argument *indent* may need to be increased.

- .NL Start a numbered list, each item preceded by a sequential number and trailing dash.
- .LI List item: This command must precede ****each**** item in the list.
- .LE End of the list (causes the indenting of the list to be cancelled).

The following text makes a four line page with the second line double size, plotted in bold face and centered.

EXAMPLES:

```
.C 1
Slides Made With VU Are
.C 2
.L 2
.B
.CE
VERY SEXY
.R
.C 3
And Also SIMPLE To Make
.C 4
QUICK and INEXPENSIVE
```


Slides Made With VU Are

VERY SEXY

And Also SIMPLE To Make

QUICK and INEXPENSIVE

window: Window a Time Series

`window(x, start, end)`

ARGUMENTS:

- x:** time series. Missing values (NAs) are allowed.
- start:** optional new starting date for the series. If earlier than start date of *x*, no change is made.
- end:** optional new ending date for the series; if later than end date of *x*, no change is made.

VALUE:

a time series like *x* giving the data between the new *start* and *end* dates.

EXAMPLES:

```
sgnp—window(gnp,c(1970,1),c(1975,12))  
#subseries of gnp from Jan 1970 through Dec 1975
```

write: Write Data to File

```
write(data, file, ncol)
```

ARGUMENTS:

data: vector.

file: optional character string naming the file on which to write the data. The name of *data* will be used by default; if *data* has no name, the file will be called "data".

ncol: optional number of data items to put on each line of *file*. Default is 5 per line for numeric data, 1 per line for character data.

Because of the way matrix data is stored, matrices are written in column by column order.

See also functions *dput*, and *dump*.

EXAMPLES:

```
write(x,"filex")
```

```
write(t(x),"byrows") # write matrix row by row
```

xor: Exclusive Or

```
xor(arg1, arg2)
```

ARGUMENTS:

arg1: logical vectors to be combined by means of an exclusive-or operation. Missing values (NAs) are allowed.

VALUE:

logical result of exclusive-or applied to *arg1* and *arg2*.

Topic - Documentation Index

Topic - Documentation Index

anova	twoway
apply*	apply, sapply, sweep
array*	aperm, apply, array, backsolve, cancel, cbind, chol, col, cutree, diag, discr, dist, eigen, hclust, l1fit, leaps, loglin, matrix, mdscale, mstree, mulbar, ncol, plclust, plotfit, prcomp, rbiwt, reg, regress, regsum, rreg, scale, solve, subtree, svd, sweep, t, twoway
boolean	logic, logicop, xor
category	code, cut, loglin, split, table, tprint
chapters	chapter, search
clustering	cutree, hclust, labclust, plclust, subtree
combinatorial	napsack
contingency table	loglin, table, tprint
data	structur
management	assign, detach, dget, dput, dump, extract, get, list, prefix, restore, rm, save, search
deferred graphics	BPLOT, defer, replot
device*	hp72, hpgl, photo, printer, show, tek12, tek46

distribution	beta, cauchy, chisq, f, gammadist, lnorm, logis, norm, ppoints, qqplot, stab, tdist, unif
editing	edit, medit
files	dump, read, restore, sink, source, write
graphic input	identify, rdpen
graphical parameters	par, pardump
hierarchical clustering	cutree, hclust, labclust, pldust
histogram	barplot, density, hist, stem
input	dget, read, restore
interpolation	approx, interp
linear algebra	backsolve, chol, eigen, gs, solve, svd
logical	all, logic, logicop, na
macros	define, macros, medit, mprint, mprompt
matrix	cbind, diag, matplot, ncol, scale, smatrix, t, tsmatrix, twoway

multiply matp, matpt

missing na

multi-dimensional scaling mdscale, mstree

multivariate cancel, cutree, discr, faces, hclust, loglin, mdscale, mstree, pairs, prcomp, smatrix, stars, subtree

normal norm

output dput, dump, write

plot* abline, barplot, box, boxplot, BPLOT, bxp, chull, contour, defer, density, devices, faces, frame, hardcopy, hatch, hist, identify, interp, labclust, lines, logo, lowess, matplot, mtext, mulbar, pairs, par, pardump, persp, pie, plclust, plot, plotfit, pretty, qqplot, range, rdpen, replot, segments, show, smatrix, stars, starsymb, symbols, text, title, tsplot, usa, vu

printing dput, print, tprint

probability beta, cauchy, chisq, f, gammadist, lnorm, logis, norm, ppoints, qqplot, tdist, unif

quantile beta, cauchy, chisq, f, gammadist, lnorm, logis, norm, ppoints, qqplot, tdist, unif

random cauchy, chisq, f, gammadist, lnorm, logis, norm, sample, stab, tdist, unif

read dget, read, restore

Topic - Documentation Index

regression llfit, leaps, rbiwt, reg, regress, regsum, rreg

robust lowess, mean, median, rbiwt, rreg, smooth, twoway

rounding ceiling, round

scaling mdscale, mstree, scale

smoothing lowess, smooth

sorting order, rank, sort

special assign, logic, operator, preceden, select, structur, syntax, System

structures compname, cstr, mode, mstr, ncomp, select, split, structur

text encode

trig acos, atan, cos

ts* diff, end, lag, smooth, time, ts, tsmatrix, tsplot, window

UNIX BATCH, BPLOT, CHAPTER, nproc, PROGRAM, PROMPT, System

utilities BATCH, BPLOT, CHAPTER, extract, PROGRAM, PROMPT

write

dput, dump, write

System Macros

Detailed Documentation

?adjust: Adjusted Variables for Regression

```
?adjust(x, y, which, regargs, ...)
```

ARGUMENTS:

x: matrix of independent variables.

y: dependent variable.

which: the column number of the variable in matrix *x* to be adjusted for the other *x* variables.

regargs: other regression arguments, i.e., arguments to the function *reg* used to control intercept (int=FALSE), weights (w=weights), etc.

VALUE:

structure with components *x* and *y*.

x: values of column *which* of the matrix *x* which are not explained by a linear fit on the other columns of *x*.

y: values of the dependent variable *y* which are not explained by a linear fit on the other *x* variables.

EXAMPLES:

```
xy ← ?adjust(myx, myy, 3)
plot(xy,xlab="Adjusted values of 3rd Variable",
     ylab="Adjusted response")
```

?apply: Apply a Macro to a Number of Arguments

```
?apply(macro, arg1, arg2, ...)
```

ARGUMENTS:

macro: the name of a macro.

argi: arbitrary list of arguments that are to be given in turn to *macro*.

EXAMPLES:

If the macro *analyze* performs an analysis of a dataset, then

```
?apply(analyze,ds1,ds2,ds3,ds4,ds5)
```

is equivalent to typing

```
?analyze(ds1)
```

```
?analyze(ds2)
```

```
...
```

?bartable: Produce Bar Plot of a Category

```
?bartable(category)
```

ARGUMENTS:

category: a category dataset, i.e., one created with *cut*, or *code*.

EXAMPLES:

```
x ← code(occupation) # create category
bartable(x) # bar plot of numbers in each occupation
```


?cleanup: Selectively Remove Datasets

?cleanup(pattern)

ARGUMENTS:

pattern: (unquoted) pattern that describes which datasets should be targeted to remove. Default is *\$T**.

EFFECT:

The name of each dataset found is printed, and a reply of *yes* causes the dataset to be deleted; *no* leaves the dataset alone.

EXAMPLES:

?cleanup # get rid of all macro temporary datasets

?col: Apply a Function to the Columns of a Matrix

?col(x, fun, arg1, ...)

ARGUMENTS:

x: name of matrix.

fun: (unquoted) name of function to be applied to the columns of the matrix.

argi: any other arguments that should be given to *fun*.

VALUE:

vector or matrix of results. If each invocation of *fun* produces a vector of length *n* ($n > 1$), the result will be an *n* by *ncol(x)* matrix. Otherwise, the result will be a vector of length *ncol(x)*.

See also function *apply*.

EXAMPLES:

?col(y,median) # column medians of matrix y

?col(y,mean,trim=.25) # 25% trimmed column means

?comment: Add a Comment to a Dataset

?comment(x,comment)

ARGUMENTS:

x: a dataset name.

comment: (unquoted) character comment to be attached to *x*.

Component *comment* is added to *x*. The comment will be printed whenever *x* is printed.

EXAMPLES:

?comment(gnp,Gross national product from government figures)

?Cp: All Subsets Regression and Cp Plot

`?Cp(x, y, wt, int, names, identify)`

ARGUMENTS:

x: matrix of possible independent variables.
y: vector of dependent variable.
wt: optional vector of weights for each observation.
int: logical flag, should intercept term appear in regression equations. Default TRUE.
names: optional character vector of names for the variables. Default is to label the columns of *x* (variables) with successive letters from the set "123456789ABCDEFGH...Z".
identify: logical flag, should user identify regressions from plot which will then be used to produce full regression statistics? Default TRUE.

EFFECT:

Produces Cp plot and full regression statistics for regressions optionally selected from the plot.

EXAMPLES:

`?leaps(variables,response) # all-subsets regression and Cp plot`

?csweep: Sweep Out Column Effects from Matrix

`?csweep(x, summary, fun)`

ARGUMENTS:

x: matrix.
summary: values to be swept out of the columns of *x*. Generally *summary* is the result of a `?col` macro or applying a function to the columns of *x*.
fun: (unquoted) name of function to be used in sweeping out *summary*. Default is - (minus).

VALUE:

matrix *x* with the summary values swept out.

EXAMPLES:

`?csweep(x, ?col(x,median) ) # subtract off column medians`

?define: Define a Macro at Execution Time

`?define(name,definition)`

ARGUMENTS:

name: name of macro to be defined, i.e., defines macro in dataset *mac.name*.
definition: unquoted definition for the macro.

EXAMPLES:

`?define(remember,abc,def,ghi)`
 # creates dataset mac.remember that can be used
 # to remember values from one invocation of the macro
 # processor to another

?discr: Discriminant Analysis with Grouping Vector

?discr(x, group)

ARGUMENTS:

x: matrix of data for discriminant analysis.

group: vector of length *nrow(x)* telling which group each row of *x* belongs to.

VALUE:

structure like that returned by the *discr* function.

EXAMPLES:

?discr(mydata,group)

?dist2full: Change Distance Structure to Full Symmetric Matrix

?dist2full(x)

ARGUMENTS:

x: distance structure, as produced by the *dist* function.

VALUE:

full symmetric matrix containing the distances from point *i* to point *j* in the *i,j*th element.

EXAMPLES:

xx ← ?dist2full(dist(datamatrix))

?extract: Extract Columns of Data from File

?extract(name, options)

ARGUMENTS:

name: unquoted name of data file. A file of the name *name.des* must have been pre-defined, in the manner described for utility *extract*.

options: any further options to the *restore* function, e.g., pos=, print=.

EFFECT:

Columns of the file *name* are read and assigned in a database. The descriptor file determines the names of the datasets created.

EXAMPLES:

```
?extract(mydata,print=T) # extract from file mydata
# based on descriptor file mydata.des
# print names of extracted datasets as they are
# placed on the work file
```

?full2dist: Turn Full Symmetric Matrix into Distance Structure

`?full2dist(x)`

ARGUMENTS:

x: square symmetric matrix.

VALUE:

structure like that produced by the *dist* function, containing components *Data* and *Size*, and containing just the lower-triangle of data from *x*.

EXAMPLES:

```
mydist ← ?full2dist(1-cor(x,x)) # turn 1 - correlation of  
# cols of x into distance structure
```

?idplot: Scatter Plot with Identifiers for Each Point

`?idplot(x, y, xlab=, ylab=, cex=)`

ARGUMENTS:

x: x-coordinates of data points.

y: y-coordinates of data points.

xlab=: unquoted label for x axis, default is the expression used for *x*.

ylab=: unquoted label for y axis, default is the expression used for *y*.

cex=: character size to be used to plot the point identification numbers. Default is 1.

EFFECT:

Produces a scatterplot, but with the points on the plot labelled by 1,...,n.

EXAMPLES:

```
?idplot(sqrt(gnp),log(income))
```

?intersect: Intersection of Two Lists

`?intersect(x, y)`

ARGUMENTS:

x: vector containing one list.

y: list.

VALUE:

vector of (unique) values which appear in both lists.

?listall: List Contents of all Current Databases

?listall(pattern)

ARGUMENTS:

pattern: optional character string giving a pattern of dataset names to be listed.

EFFECT:

Prints each database name, followed by a listing of its contents. If a prefix is in effect, only datasets within the prefix are listed.

EXAMPLES:

?listall

?listall("mac.*") # list macros on all databases

?listfun: List Names of all S Functions Currently Available

?listfun()

VALUE:

character vector containing names of all S functions accessible on any of the current chapters. Names of the form "x0..." are internal names of operators such as "+", etc.

EXAMPLES:

?listfun # get list of all functions

?mat2vecs: Change Columns of Matrix to Individual Vectors

?mat2vecs(x, names)

ARGUMENTS:

x: matrix of data.

names: character vector of names for the columns of x. Should not contain embedded blanks, special characters, etc.

VALUE:

This function creates on the work file datasets named by *names*, one for each column of x.

EXAMPLES:

?mat2vecs(data,c("age","height","weight")) # creates 3 vectors
named age, height, and weight

?mdsplot: Plot the Results of Multidimensional Scaling

?mdsplot(mds)

ARGUMENTS:

mds: matrix representing data points, such as that produced by the function *mdscale*.

EFFECT:

Produces a plot where the x and y axes have the same coordinates per inch, thus distances are represented correctly. Each point is identified by its sequence number.

EXAMPLES:

?mdsplot(m ← mdscale(distmat))

?medit: Edit a Macro

?medit(name)

ARGUMENTS:

name: unquoted name of macro.

EFFECT:

Prepends *\$mac.* to *name* and invokes the *medit* function. This makes sure that macro names are correct even when a prefix is in effect.

EXAMPLES:

?medit(probsamp) # edit the probsamp macro

?missing: Find Observations Containing NAs

?missing(x, y)

ARGUMENTS:

x: matrix, typically the x matrix for a regression.

y: vector with *nrow(x)* values.

VALUE:

logical vector with *nrow(x)* values, where TRUE indicates a missing value in the corresponding row of x or value of y.

EXAMPLES:

i ← ?missing(x,y)
regress(x[!i], y[!i])

?mlist: List Names of Macros

?mlist(pos)

ARGUMENTS:

pos: position on database search list of database containing macros. Default is 2.

VALUE:

character vector of macro names.

EXAMPLES:

?mlist # names of all macros on user's database

?mlist(pos=3) # names of macros on system database

?pairs: All Pair-wise Scatter Plots

?pairs(x, labels, head)

ARGUMENTS:

x: matrix.

labels: character vector of names for the columns of x.

head: unquoted character string to be used to title the page of scatter plots.

EFFECT:

This macro is similar to the *pairs* function; however the macro produces all pair-wise scatter plots with labelled axes. All plots will appear on a single page, therefore there should not be too many columns to the matrix x.

EXAMPLES:

?pairs(longley.x,label=longley.collab)

?probsamp: Sample of Data with Specified Probability

?probsamp(x, alpha)

ARGUMENTS:

x: vector of data.

alpha: probability (between 0 and 1) of picking any individual data value in x to be in the generated sample. If *alpha* is not given, its user is prompted for the value.

VALUE:

a sample of the data vector x.

See also function *sample*.

EXAMPLES:

?probsamp(mydata,.5) # return sample of mydata

each value in mydata has 50% chance of appearing in sample

?prompt: Create Documentation for Macros or Datasets

?prompt(name, pos=)

ARGUMENTS:

name: the name of the macro or dataset(s) to be documented. If there is no macro named *name*, dataset documentation for *name* is produced. If there are several datasets whose names begin with *name*, they will all be included in the documentation file.

pos=: database number on which the macro or dataset(s) reside. Default is 2.

EFFECT:

Creates file *name.d* with shell of the documentation.

EXAMPLES:

?prompt(abc) # documentation for macro or dataset abc

?prompt(longley,pos=3) # documentation for longley data on system database

?qqplot: Generate Probability Plot for Arbitrary Distribution

?qqplot(x, dist,par1,par2)

ARGUMENTS:

x: Data to be plotted.

dist: Name of distribution (e.g., t, beta, chisq) Default is ?PROMPT(What distribution?).

pari: Optionally, parameters to the distribution. (Note that location and scale parameters are not needed; only shape or degrees of freedom).

EFFECT:

Produces a probability (Q-Q) plot from an arbitrary distribution, so long as there is a corresponding quantile function.

EXAMPLES:

?qqplot(rnorm(50), t, 3) # random normals vs t on 3 d.f.

?ranktest: Two-sample Rank Test

?ranktest(x, y)

ARGUMENTS:

x: vector of data from first sample.

y: vector of data from second sample.

VALUE:

both datasets are ranked together, and the sum of the ranks of the values in the first set is returned.

EXAMPLES:

?ranktest(heights1,heights2)

?row: Apply a Function to the Rows of a Matrix

?row(x, fun, arg1, ...)

ARGUMENTS:

x: name of matrix.
fun: (unquoted) name of function to be applied to the rows of the matrix.
argi: any other arguments that should be given to *fun*.

VALUE:

vector or matrix of results. If each invocation of *fun* produces a vector of length n ($n > 1$), the result will be an $nrow(x)$ by n matrix. Otherwise, the result will be a vector of length $nrow(x)$.

See also function *apply*.

EXAMPLES:

```
?row(y,median) # row medians of matrix y
?row(y,mean,trim=.25) # 25% trimmed row means
```

?rperm: Random Permutation

?rperm(x)

ARGUMENTS:

x: vector of data.

VALUE:

random permutation of the values in *x*.

See also function *sample*.

EXAMPLES:

```
?rperm(mydata)
```

?rsweep: Sweep Out Row Effects from Matrix

?rsweep(x, summary, fun)

ARGUMENTS:

x: matrix.
summary: values to be swept out of the rows of *x*. Generally *summary* is the result of a *?row* macro or applying a function to the rows of *x*.
fun: (unquoted) name of function to be used in sweeping out *summary*. Default is - (minus).

VALUE:

matrix *x* with the summary values swept out.

EXAMPLES:

```
?rsweep(x, ?row(x,median) ) # subtract off row medians
```

?sample: Random Sample

?sample(x, n)

ARGUMENTS:

x: vector of data.

n: number of values from x to be included in the sample. Default is to prompt the user for the desired sample size.

VALUE:

sample of *n* values randomly chosen from *x*.

See also function *sample*.

EXAMPLES:

?sample(mydata,10) # chose sample of 10 values from mydata

?ssweep: Sweep Out Effects from Structure

?ssweep(x, summary, fun)

ARGUMENTS:

x: structure.

summary: values to be swept out of the components of *x*. Generally *summary* is the result of the *sapply* function.

fun: (unquoted) name of function to be used in sweeping out *summary*. Default is - (minus).

VALUE:

structure *x* with the *summary* values swept out.

EXAMPLES:

?ssweep(x, sapply(x,"median")) # subtract off medians
from vector in each component in x

?union: Union of Several Lists

?union(x1, x2, ...)

ARGUMENTS:

xi: vector containing a list.

VALUE:

vector of the (unique) values appearing in at least one of the lists.

?vecs2mat: Make Matrix from a Set of Vectors

```
?vecs2mat(name, arg1, arg2, ...)
```

ARGUMENTS:

name: name to be used for output.

argi: data vectors to be made into the columns of constructed matrix.

EFFECT:

The matrix produced will be called *name.data*, and the character vector containing column names will be called *name.collab*.

EXAMPLES:

```
?vecs2mat(mymat,height,weight,age) #create 3-column matrix  
# mymat.data and vector of column labels mymat.collab
```

?which: Find Which Observations Satisfy a Condition

```
?which(condition)
```

ARGUMENTS:

condition: logical expression.

VALUE:

vector of subscripts that tell which observations satisfy *condition*.

EXAMPLES:

```
?which(age<30 & height>72) # list of observation numbers  
# for tall people under 30
```

System Datasets

Detailed Documentation

Note: Where more than one name appears under a heading (e.g., *number* and *payoff* under *lottery*), the actual dataset names will be composed of the heading, a period, and the name (for example, *lottery.payoff*).

akima: Waveform Distortion Data for Bivariate Interpolation

x,y,z: represents a smooth surface of z values at selected points irregularly distributed in the x-y plane.

SOURCE:

Hiroshi Akima, "A Method of Bivariate Interpolation and Smooth Surface Fitting for Irregularly Distributed Data Points", *ACM Transactions on Mathematical Software*, Vol 4, No 2, June 1978, pp 148-159.

Hiroshi Akima, "A method of Bivariate Interpolation and Smooth Surface Fitting Based on Local Procedures", *CACM* Vol 17, No 1, pp 18-20.

author: Character Counts for Books by Various Authors

count: matrix of 26 columns corresponding to letters of the alphabet. Each row contains data for one work, giving the counts of each letter. There are different total counts for each work, and proper nouns were removed prior to counting.

collab: column labels, the letters of the alphabet.

rowlab: row labels, book titles and author names.

auto: Statistics of Automobile Models.

collab: names of the columns (variables). These include price, dimensions, repair record (coded on a 5-point scale, large==good), turning radius engine displacement and turning radius.

rowlab: names of the automobile models summarized.

stats: actual statistics corresponding to the variables and models. The matrix gives data on 74 models by 12 statistics.

The information concerns automobiles of the 1979 year as offered in the United States.

SOURCE:

Various sources, primarily *Consumer Reports*, April 1979, and United States government EPA statistics on fuel consumption.

bicoal: Bituminous Coal Production in USA

tons: production in millions of net tons per year, 1920-1968.

From Tukey, *Exploratory Data Analysis*, p212.

bonds: Daily Yields of 6 AT&T Bonds

yield: matrix of daily bond yields; rows represent 192 days from April 1975 to December 1975.

Columns are one of 6 bonds, characterized by their coupon rate.

coupon: vector of 6 different coupon rates corresponding to columns of *yield*.

collab,label: Column and overall label.

car: Fuel Consumption Data

time: number of days since initial car purchase.

miles: number of miles driven between this fillup and previous fillup.

gals: number of gallons required to fill tank.

This data pertains to a new, 1974 model automobile. Source, R. A. Becker, personal data.

cereal: Consumer Attitudes Towards Breakfast Cereals

attitude: matrix giving percentage of people agreeing with 11 statements (rows) about 8 brands of cereals (columns).

rowlab: statements about cereals, i.e., "Reasonably Priced", etc.

collab: brand of cereal.

SOURCE:

A. S. C. Ehrenberg, "Graphs or Tables", *Conference on Graphical Methods*, Sheffield, March 1977.

Taken from T. K. Chakrapini and A. S. C. Ehrenberg, "Factor Analysis or BGA?", Paper presented to Multivariate Study Group of RSS, February 2, 1976.

chernoff2: Mineral Contents Data (used by Chernoff)

The data is a 53 by 12 matrix representing mineral analysis of a 4500 foot core drilled from a Colorado mountainside. Twelve variables (columns) represent assays of seven mineral contents by one method and repeated assays of five of these by a second method. Fifty-three equally spaced specimens (rows) along the core were assayed. Specimen ID numbers were 200 to 252.

SOURCE:

H. Chernoff, "The use of Faces to Represent Points in k-dimensional Space Graphically", *J. Am. Stat. Assn.* 68(1973), p361-368.

city: Names and Location of Selected US Cities

name: character vector of city names.

state: character vector giving state for each city.

x,y: location of city: x is negative longitude to correspond to coordinate system set up by function *usa*. y is latitude (both measured in degrees).

These cities cover the geographical regions of the US, and were not chosen by population.

co2: Mauna Loa Carbon Dioxide Concentration

This data represents monthly CO2 concentrations from January 1958 to December 1975.

SOURCE:

Bert Rust, Oak Ridge Associated Universities, Presented at Conference on Numerical and Statistical Computing, June 20-21, 1977, Stanford University.

corn: Corn Yields and Rainfall

yield,rain: yearly corn yield and rainfall measurements (Midwest US?) from 1890 to 1927.

Source: Ezekiel and Fox, *Methods of Correlation and Regression Analysis*, p 212.

evap: Soil Evaporation Data

collab: column labels for x.

x: matrix of independent variables: maximum daily soil temperature, minimum daily soil temperature average soil temperature (integrated area under daily soil temperature curve), maximum, minimum, and average air temperature, maximum, minimum, and average relative humidity, and total wind (miles per day).

y: daily amount of evaporation from the soil.

"It is desired to estimate the daily amount of evaporation from the soil as a function of air temperature, relative humidity, and wind. Since these factors vary considerably throughout the day it is not clear what function or aspect of these variables is most important. For this reason the following ten variables relating to these factors are recorded." (from Freund).

Observations represent 46 consecutive days from June 6 through July 21.

SOURCE:

R. J. Freund, "Multicollinearity Etc., Some "new" Examples", *Amer. Stat. Assn. Proc of Statistical Computing Section*, 1979, pp 111-112.

freeny: Revenue Data

y: quarterly revenue, 39 observations from (1962,2Q) to (1971,4Q).
x: matrix of independent variables, Columns are y lagged 1 quarter, price index, income level, and market potential.

SOURCE:

A. E. Freeny, "A Portable Linear Regression Package with Test Programs", Bell Laboratories memorandum.

hstart: US Housing Starts

US Housing Starts, monthly. Source unknown.

iris: Fisher's Iris Data

Array giving 4 measurements on 50 flowers from each of 3 varieties of iris. The measurements are sepal length and width and petal length and width. Varieties are Setosa, Virginica, and Versicolor.

SOURCE:

R. A. Fisher, *Ann. Eugen.*, Vol 7 (1936), 179-188.

liver: Carcinogenicity Studies of Rat Livers

cells: number of cells injected into each animal.
exper: category for the three experiments (A, B, C) in the study.
gt: matrix (52 by 4) for the 4 lobes (ARL, PRL, PPC, AC), with counts for each (observation,lobe) pair of the GT(+) colonies.
section: category for the section (replication) for each observation. Successive observations are pairs of sections for the same specimen.

SOURCE:

"Detection of Damaged Animals in Carcinogenicity Studies with Laboratory Animals", Camil Fuchs (1980), Univ. of Wisconsin Statistics Laboratory Report 80/3.

longley: Longley's Regression Data

y: number of people employed, yearly from 1947 to 1962.
x: matrix with 6 columns, giving gnp deflator, gnp, unemployed, armed forces, population, and year.

This regression is known to be highly collinear.

SOURCE:

J. W. Longley, "An appraisal of least-squares programs from the point of view of the user", *J. Amer. Statist. Assoc.*, 62 (1967) 819-841.

lottery: New Jersey Pick-it Lottery Data

number: winning 3-digit number (from 000 to 999) for drawings from May 22, 1975 to Mar 16, 1976.
(This was the beginning of the Pick-it lottery).
payoff: payoff corresponding to each *number* (dollars).

lottery2: New Jersey Pick-it Lottery Data (Second Set)

number: winning 3-digit number (from 000 to 999) for drawings from Nov 10, 1976 to Sep 6, 1977.
payoff: payoff corresponding to each *number* (dollars).

lynx: Canadian Lynx Trappings

The data give annual number of lynx trappings in the Mackenzie River District of North-West Canada for the period 1821 to 1934.

SOURCE:

Elton and Nicholson, "The ten-year cycle in numbers of lynx in Canada", *J. Anim. Ecol.*, 11(1942), 215-244.

Analyzed in: M. J. Campbell and A. M. Walker, "A Survey of Statistical Work on the Mackenzie River Series of Annual Canadian Lynx Trappings for the Years 1821-1934 and a New Analysis", *J. R. Statist. Soc. A*, 140 Part 4 (1977), 411-431.

month: Month Names and Abbreviations

name,abb: character name and abbreviations for months of the year.

nytel: Expenditures and Growth for New York Telephone

main: main telephone and equivalent main telephone gain in millions for years 1960-1971.
constr: total growth construction expenditures (1967 millions of dollars).

ozone: Ozone Concentrations in North-East US

median,quartile: median and upper quartile of daily maxima ozone concentration for June-August, 1974. Concentrations in ppb.
city: character name of ozone monitoring site.
xy: structure containing components named *x* and *y*, that give the negative longitude and altitude of the monitoring sites (in the coordinate system used by function *usa*).

SOURCE:

Cleveland, Kleiner, McRae, Warner, Pasceri, "The Analysis of Ground-Level Ozone Data from New Jersey, New York, Connecticut, and Massachusetts: Data Quality Assessment and Temporal and Geographical Properties", Bell Laboratories Memorandum, July 17, 1975.

prim4 prim9: Particle Physics Data

prim4 is a matrix of 500 experimental observations, with 4 parameters describing each. *prim9* is a similar matrix, but with 9 parameters describing each observation.

SOURCE:

Examples produced from work on the Prim-9 graphics system at Stanford Linear Accelerator. Probably related to the data cited in J. H. Friedman and J. W. Tukey, *I.E.E.E. Trans. Comp.*, vol. c-23, 881- 889. If so, these are particle-scattering experiments, and the columns of the matrices represent some chosen set of parameters (energy, momentum, etc.) to describe the nuclear reactions.

rain: New York City Precipitation

nyc1: New York City precipitation, 1869-1957, currently listed in World Weather Records.

nyc2: New York City precipitation, 1869-1957, formerly listed in World Weather Records and found in some almanacs.

SOURCE:

J. W. Tukey, *Exploratory Data Analysis*, Addison-Wesley (1977), p206. The value for 1946 was read off the plot.

Random.seed: Seed Values for Random Number Generators.

A vector of starting values for the various random number generators used in S. The generators (for all distributions) are organized so that successive random numbers are equivalent to a long sample from the underlying uniform distribution. This allows the long-term properties of the generator to be maintained. The first time a user generates some random numbers, the dataset *Random.seed* is found on the system database. The values contained are the standard initial values of the underlying generator algorithm. After each random sample, the dataset *Random.seed* is assigned on the users working database, to maintain consistent values on subsequent calls.

There is a useful technique for reproducing random samples in later work. Just copy *Random.seed* before generating the sample for the first time, and then restore it when the sample is to be reproduced.

EXAMPLES:

```
oldseed ← Random.seed #save it
y ← rnorm(1000) # get sample, analyse, etc.
...
Random.seed ← oldseed #restore seed
yagain ← rnorm(1000) # will be the same as y
```

Source: G. Marsaglia, et al. *Random Number Package: "Super-Duper"*, School of Computer Science, McGill University, 1973. (Our generator, *uni* is a portable version of their generator.)

saving: Savings Rates for Countries

x: matrix with 50 rows (countries) and 5 columns (variables)
rowlab: country names corresponding to rows of x.
collab: names of the variables (columns) in x.

SOURCE:

Used by D. C. Hoaglin and R. E. Welsch, "The Hat Matrix in Regression and ANOVA", (1976), Memorandum NS-341, Dept. of Statistics, Harvard University. Originally from unpublished data of Arlie Sterling, these are averages over 1960-1970. Income is per capita disposable income in U.S. dollars (as of 1970?); growth is the per cent growth of income.

ship: Manufacturing Shipments

Source: Unknown.

stack: Stack-loss Data

This data represents 21 days of operation of a plant oxidizing ammonia to get nitric acid.

loss: percent of ammonia lost (times 10).
x: matrix with 21 rows and 3 columns representing air flow, cooling water inlet temperature, and acid concentration.
collab: character labels for columns of x.

SOURCE:

Brownlee, *Statistical Theory and Methodology in Sci. and Eng.*, (1965). Also in Draper and Smith, *Applied Regression Analysis*, Ch 6, and Daniel and Wood, *Fitting Equations to Data*, (1971), p61.

state: States of the US

name,abb: character names and abbreviations for 50 US states, sorted alphabetically by name.
center: structure with components named x and y giving the approximate geographic center of each state in negative longitude and latitude (as used by function *usa*). Alaska and Hawaii are placed just off the West Coast.
region: category giving region (Northeast, South, North Central, West) that each state belongs to.
division: category giving state divisions (New England, Middle Atlantic, South Atlantic, East South Central, West South Central, East North Central, West North Central, Mountain and Pacific).
x77: matrix giving statistics for the states for 1977. Columns are Population, Income, Illiteracy, Life Expectancy, Murder Rate, Percent High-school Graduates, Days of Frost, and Area.
col77: character vector of column labels for x77 matrix.

swiss: Fertility Data for Switzerland in 1888

fertility: standardized fertility measure I/g for each of 47 French-speaking provinces of Switzerland at about 1888.

x: matrix whose columns give 5 socioeconomic indicators for the provinces: 1) percent of population involved in agriculture as an occupation. 2) percent of "draftees" receiving highest mark on army examination. 3) percent of population whose education is beyond primary school. 4) percent of population who are Catholic. 5) percent of live births who live less than 1 year: infant mortality.

collab: short column labels for x.

SOURCE:

Mosteller and Tukey, *Data Analysis and Regression*, Addison Wesley, 1977, p549-551.

switzerland: Heights of Switzerland on 12 by 12 Grid

Data gives height in 1000's of feet to nearest 1000?

Source: Unknown.

telsam: Interviewer Response Data

response: table giving counts of number of answers poor, fair, good, and excellent (the columns) for a number of different interviewers (the rows).

collab: character label for columns of *response*.

rowlab: interviewer identification number.

tone: Bricker's Tone-Ringer Preference Data

appeal: matrix giving standardized ratings of 100 tones (rows) by 43 subjects (columns). Standardized so that each subject has mean 0 and biased variance 1.

rowlab: character identifiers for tones.

SOURCE:

P. J. Bricker, *Bell Syst. Tech. J.*, (1971), 1559-1578.

util: Earnings and Market/Book Ratio for Utilities

earn: earnings of 45 utilities.

mktbook: market price to book value ratio for the utilities.

votes: Votes for Republican Candidate in Presidential Elections

repub: percent of votes given to republican candidate in presidential elections from 1856 to 1976.
Rows represent the 50 states (See dataset *state.*), and columns the 31 elections.

year: year of each election (corresponds to columns of *x*).

Contains missing values (NAs) for years prior to statehood.

SOURCE:

Peterson, S. (1973) *A Statistical History of the American Presidential Elections*, Ungar, New York.
Data from 1964 to 1976 is from Scammon "American Votes", *Congressional Quarterly*, vol. 12.

Algorithm Detailed Documentation

abort: abort execution

subroutine abort

adjust: proportional adjustment of table

algorithm as 51.2 appl. statist. (1972), vol 21, no 2
subroutine adjust(nvar,x,y,z,locx,locy,locz,nx,ny,nz,dim,config,d)
integer size(16),dim(nvar),config(nvar),coord(15)
real x(nx),y(ny),z(nz)

aincr: Incremental Counter for Array Index

subroutine aincr(ind,ext,k,flag)
integer ind(k),ext(k),k,flag
integer j

alngam: $\log(\text{abs}(\text{gamma}(x)))$

real function alngam(x)
june 1977 edition. w. fullerton, c3, los alamos scientific lab.
real x,sinpiy,sq2pil,sqpi2l,pi,xmax,dxrel,y
real gamma,r9lgmc

applrf: apply reflector

subroutine applrf(c,s,k,l,x,y,jstepx,jstepy)
integer k,l,i,jx,jy,jstepx,jstepy,iter
real c,s,x(1),y(1)
replaces $x[k:l], y[k:l]$ by $(x[k:l], y[k:l])G$ where G is the Givens
matrix determined by c and s. Jstepx and jstepy are the respective
step sizes.

betai: probability variable from beta less than x

```
real function betai(x,pin,qin)
real x,pin,qin
# april 1977 version. w. fullerton, c3, los alamos scientific lab.
# based on bosten and battiste, remark on algorithm 179, comm. acm,
# v 17, p 153, (1974).
```

calcrf: calculate reflector

```
subroutine calcrf(x,y,c,s)
real x,y,c,s,t,u,v,mu
```

cfill: fill character string with character

```
subroutine cfill(value,x,n)
CHARACTER(value,l)
CHARACTER(x,*)
integer n,i
```

ch2vec: Convert packed characters into vector

```
POINTER function ch2vec(chars,n,leng)
# converts packed characters into vector of character
# strings on stack - returns ptr to vector of ptrs
# strips trailing blanks
CHARACTER(chars,*)
integer n,leng
```

chash: hash table search for char. string

```
integer function chash(item,tab,tlen,enter)
integer tlen; POINTER item,tab(tlen); logical enter
# returns: i>0 if lookup successful or if enter successful
# -i if enter and already present
# 0 if not there
# tlen+1 if table full
```


chcomp: compare (order) 2 char strings given pointers

```
# note that this routine would be needed for sortf on chars
integer function chcomp(p1,p2)
POINTER p1,p2
```

chol: choleski decomposition

```
subroutine chol(t,p,l,singlr)
integer p
real t(p,p),l(p,p)
logical singlr
```

collap: compute or remove margin from table

```
# algorithm as 51.1 appl. statist. (1972), vol 21, no 2
subroutine collap(nvar,x,y,locx,locy,nx,ny,dim,config,option)
# if option is true, computes a marginal table from a complete
# table. if option is false, removes an effect from a table.
integer size(16),dim(nvar),config(nvar),coord(15)
logical option
real x(nx),y(ny) #the larger table is x and the smaller one is y
```

coment: put comment into structure

```
subroutine coment(msg,str)
CHARACTER(msg,*); POINTER str
```

concat: copy string into buffer

```
subroutine concat(buf,istart,from,fstart,length)
CHARACTER(buf,*); CHARACTER(from,*)
integer istart, fstart, length
```

corssd: correlations by SSD formula

```
real function corssd(x,y,n,trm,scr)
integer n,ind(2),i
real x(1),y(1),trm,scr(n,2),v1,v2,trmean,trvar,xm,ym,xs,ys
```

csevl: evaluate chebyshev series cs at x

```
real function csevl(x,cs,n)
integer n
real x,cs(n)
# april 1977 version. w. fullerton, c3, los alamos scientific lab.
# evaluate the n-term chebyshev series cs at x. adapted from
# r. broucke, algorithm 446, c.a.c.m., 16, 254 (1973). also see fox
# and parker, chebyshev polys in numerical analysis, oxford press, p.56.
#
#      input arguments --
# x      value at which the series is to be evaluated.
# cs     array of n terms of a chebyshev series. in eval-
#        uating cs, only half the first coef is summed.
# n      number of terms in array cs.
```

ctype: integer class of a character

```
integer ctype(c)
CHARACTER(c,*)
#returns one of the classes in $I/ctype
```

ddotr: double precision dot product of real array sections

```
double precision function ddotr(c,a,istepa,n,b,istepb)
integer n,istepa,istepb,i,ia,ib
real a(n),b(n),c
double precision sum
```

deref: dereference an entry

```
POINTER function deref(pp)
POINTER pp
```


dev: deviations from column means

```
subroutine dev(x,n,ndim,p,devs,means)
integer n,ndim,p; real x(ndim,p),devs(ndim,p),means(p)
# x:    n by p matrix of data, dimensioned with ndim rows
# devs: matrix, similar to x, for deviations (can be x)
# means: vector for column means
```

devw: remove x and y means given (sqrt) weights

```
subroutine devw(x,n,k,y,ky,xc,yc,q)
integer n,k,ky,j; real x(n,1),y(n,1),xc(1),yc(1),q(1),rho
```

dget: get a data set from file

```
POINTER function dget(dummy)
integer dummy
```

digit: numeric value of character

```
integer function digit(c)
CHARACTER(c,1)
```

dirclp: collapse directories (removes USED entries)

```
subroutine dirclp(dir)
POINTER dp,pn,dir
```

dirfnd: find entry in directory

```
given name
POINTER function dirfnd(p,name)
POINTER p
CHARACTER(name,*)
```

disbin: binary distance

```
real function disbin(a,da,b,db,p)      # binary (proportion) distance
real a(1),b(1),sum1,sum2
integer da,db,i,j,k,p
```

diseuc: euclidian distance

```
real function diseuc(a,da,b,db,p)      # euclidian distance
real a(1),b(1),sum
integer da,db,i,j,k,p,nact
```

disman: manhattan distance

```
real function disman(a,da,b,db,p)      # manhattan distance
real a(1),b(1),sum
integer da,db,i,j,k,p,nact
```

dismax: maximum distance

```
real function dismax(a,da,b,db,p)      # maximum distance
real a(1),b(1),sum
integer da,db,i,j,k,p,nact
```

distan: compute distance (triangular storage)

```
subroutine distan(x,n,p,d,dfun)
integer n,p
real x(n,p),d(1),dfun
external dfun
# x: input, matrix (n by p) of data
# d: output, distances stored by columns in triangular array
# dfun: function, declared external in caller, specifying which function
#      to use in calculations (see metric.r)
```


dot: general dot product of array sections

```
real function dot(c,a,istepa,n,b,istepb)
integer n,istepa,istepb,i,ia,ib
real a(n),b(n),c
double precision sum
```

dotv: dot product of 2 vectors

```
real function dotv(a,n,b)
integer n; real a(n),b(n)
double precision sum
integer i
```

dput: put a data structure onto the output file

```
subroutine dput(file,str)
integer file; POINTER str
```

dtafmt: format for vector of data

```
subroutine dtafmt(vx,mode,n,width,par1,par2)
integer mode,n,width,par1,par2; POINTER vx
```

eigen1: get M largest or smallest eigenvals & vectors

```
subroutine eigen1(A,n,m,values,vector,large)
# A: n by n **symmetric** matrix
# m: no. of eigenvalues & vectors wanted
# values: vector for output eigenvalues
# vector: n by m matrix for output eigenvectors (in columns)
# large: flag for largest(TRUE) or smallest(FALSE) eigenvalues
integer n,m; real A(n,n),values(n),vector(n,m); logical large
```

erf: error function of x

real function erf(x)
real x

erfc: complimentary error function of x

real function erfc(x)
july 1977 edition. w. fullerton, c3, los alamos scientific lab.
real x

excmd: execute system command

subroutine excmd(string,leng)
CHARACTER(string,*); integer leng

fitden: density estimation

subroutine fitden(x,n,kindow,width,first,last,nout,out)
real x(1),out(1),width,first,last
integer n,nout,kindow

fixnam: create data set name using prefix if any

POINTER function fixnam(text)
POINTER text,p
text pointer to data set name
if text starts with "\$", prefix is ignored, and "\$" stripped
otherwise, prefix, if any is prepended to text
returns, in either case, pointer to the actual data set name

gam: computes gamma(eta)

real function gam(eta)
real eta

gamlim: legal bounds for x in gamma(x)

subroutine gamlim(xmin,xmax)
real xmin,xmax

gamlog: computes naperian log of gamma(eta)

real function gamlog(eta)
real eta

gamma: returns gamma(x)

real function gamma(x)
real x

getds: get a data set

POINTER function getds(name,pos)
POINTER name; integer pos

getpth: get path name for ith directory on search list

POINTER function getpth(name,which,i)
CHARACTER(name,*)
integer which,i
POINTER path

gtds: get data structure (given file descriptor)

POINTER function gtgs(fd)
integer fd

gtline: read a line**han: hanning**

```
subroutine han(y,scr,n)
integer n
real y(n),scr(n)
```

hcmavm: cluster merge by average

```
subroutine hcmavm(d,n,iold,inew,w)
real d(1), w(1)
```

hcmmam: cluster merge by maximum

```
subroutine hcmmam(d,n,iold,inew,w)
real d(1), w(1)
```

hcmnim: cluster merge by minimum

```
subroutine hcmnim(d,n,iold,inew,w)
real d(1), w(1)
```

i2str: make string pointer from integer

```
POINTER function i2str(i)
integer i
```

iascii: turn character into ascii integer

```
integer function iascii(c,case)
CHARACTER(c,1); integer case
```


ichcom: compare two strings lexicographically

```
integer function ichcom(s1,s2)
CHARACTER(s1,*); CHARACTER(s2,*)
```

icol: remove v from columns of Q

```
subroutine icol(m,n,Q,R,k,v,ndim)
integer m,n,k,ndim
real Q(m,ndim),R(ndim,ndim),v(m)
```

icopy: copy integer data

```
subroutine icopy(from,to,length)
integer from(1),to(1),length,i
```

ientry: create an entry with name mode length value

```
POINTER function ientry(n,m,l,v)
POINTER i,n,v; integer m,l
```

ifill: fill vector of integers

```
subroutine ifill(value,x,n)
integer n,i
integer value,x(n)
```

inits: find number of terms needed in orthogonal series

```
integer function inits(os,nos,eta)
integer nos; real os(nos),eta
```

inputd: input data items

subroutine inputd(data,mode,infile,skip,eof)
POINTER data,value; integer mode,infile,skip; logical eof

islenz: length of string

integer function islenz(str)
CHARACTER(str,*)

istrng: initialize string

POINTER function istrng(chars,length)
CHARACTER(chars,*)
integer length
chars - character string to be made into dynamic string on stack
length - either no. of characters in chars or -1 (look for EOS)
0 causes creation of null string
returns pointer to string on stack (with EOS)

isum: sum of vector ix of length n

integer function isum(ix,n)
integer ix(n),n

jgetch: get character from string

character function jgetch(str,i)
CHARACTER(str,*); integer i

jputch: put character into string

subroutine jputch(str,i,char)
CHARACTER(str,*); CHARACTER(char,1); integer i

kill: kill a process

```
subroutine kill(procid)
integer procid
```

l1: fit by minimum absolute residuals

```
subroutine l1(m,n,m2,n2,a,b,toler,x,e,s)
real a(m2,n2),x(n),e(m),b(m)
# barrodale and roberts, cacm (june 1974) pp 319-320
# algorithm 478
```

l1fit: regression to minimize absolute residuals

```
subroutine l1fit(x,n,ndim,p,y,b,res,int,sar,rank,status)
real x(ndim,1),y(1),res(1),b(1),sar
real tol
logical int
```

lcopy: copy logical data

```
subroutine lcopy(from,to,length)
logical from(1),to(1)
integer length,i
```

leaplb: generate labels for leaps output

```
subroutine leaplb(regid,nreg,kx,names,label,which,id,ln)
POINTER label(1),names(1)
integer regid(1),curid,id(1),ln(1)
logical which(nreg,1)
```

leaps: subset regressions by Furnival & Wilson algorithm

```
subroutine leaps(rr,kx,nr,ndef,ibit,mbst,tol,regid,criter,ncoef,nreg,r,i)
real rr(nr,1),r(1),criter(1); integer i(1),ncoef(1),regid(1)
```

letter: recognize letters

logical function letter(c)
CHARACTER(c)

lfill: fill vector of logicals

subroutine lfill(value,x,n)
integer n,i
logical value,x(n)

lowess: locally weighted scatterplot smoother

subroutine lowess(x,y,n,f,nsteps,delta,ys,rw,res)
real x(n),y(n),ys(n),rw(n),res(n)
logical ok

lowest: lowess estimate at single x value

subroutine lowest(x,y,n,xs,ys,nleft,nright,w,userw,rw,ok)
real x(n),y(n),w(n),rw(n)
logical userw,ok

match: look up character strings in table by partial match

integer function match(name, table)
POINTER name, table(1) # table terminated by USED!

matchn: match (& partial match) two names

integer function matchn(name1, name2)
CHARACTER(name1,*)
CHARACTER(name2,*)
#returns: 1 if exact match
0 if truncation of name2 == name1
-1 otherwise

matp: multiply matrices

```
subroutine matp(a,n,ndim,p,b,bdim,q,c,cdim)
integer n,ndim,bdim,cdim,p,q
real a(ndim,1),b(bdim,1),c(cdim,1)
```

mean: arithmetic mean of vector x of length n

```
real function mean(x,n)
integer n; real x(n)
double precision ds
integer i
```

mktemp: make up string pointer to unique file name

```
POINTER function mktemp(dummy)
integer dummy
```

nacopy: copy omitting NAs

```
subroutine nacopy(from,skip,n,to,nouta)
# TO may be same as FROM
# this process is inverted by NABACK
integer n,skip,nout,nouta
real from(1),to(1)
integer end,j
```

narang: min and max of real vector containing NAs

```
subroutine narang(x,n,xmin,xmax)
real x(1),xmin,xmax
integer i,n
```

nprime: next prime larger than or equal to argument

integer function nprime(n)
integer n

numode: numeric mode

integer function numode(p)
POINTER p

orderf: stable ordering of integer array x of length n (with function)

subroutine orderf(x,n,key,fun)
integer n; integer x(n)
integer key(n)
integer fun; external fun

orderi: stable ordering of integer array x of length n

subroutine orderi(x,n,key)
integer n; integer x(n)
integer key(n)

orderr: stable ordering of real array x of length n

subroutine orderr(x,n,key)
integer n; real x(n)
integer key(n)

pbeta: probability variable from beta less than quant

real function pbeta(quant,alpha,beta)
real quant,alpha,beta

pcauc: probability variable from cauchy less than quant

real function pcauc(quant,a,b)
real quant,a,b

pcopy: copy two vectors

subroutine pcopy(from,to,length,mode)
POINTER from,to; integer length,mode

pfdis: probability variable from "f" less than quant

real function pfdis(quant,df1,df2)
real quant,df1,df2

pgamm: probability variable from gamma shape eta less than quant

real function pgamm(quant,eta)
real quant,eta

plnor: probability variable from log normal less than quant

real function plnor(quant,a,b)
real quant,a,b

plogi: probability variable from logistic less than quant

real function plogi(quant,locate,scale)
real quant,locate,scale

pnorm: probability distribution for normal family

real function pnorm(x,a,b)
real x,a,b

pnorms: probability variable from standard normal less than quant

real function pnorms(quant)
real quant

pnormt: normal probability (accurate in tail)

real function pnormt(x)
real x

prehsh: turn arbitrary value into integer for hashing

#naive version - assumes can address arb. values on stack as ints
integer function prehsh(value,mode)
POINTER value; integer mode

prtfil: output the PRINT buffer

subroutine prtfil(fd)
integer fd

psort: partially sort array a of length n conditional on ind

```
subroutine psort(a,n,ind,ni)
  real a(n)
  integer n,ind(ni),ni
c
  integer indu(16),indl(16),iu(16),il(16),p,jl,ju,i,j,m,k,ij,l
  real t,tt
```

psorti: partially sort integer array a of length n conditional on ind

```
subroutine psorti(a,n,ind,ni)
  integer a(n),n,ind(ni),ni
c
  integer indu(16),indl(16),iu(16),il(16),p,jl,ju,i,j,m,k,ij,l,t,tt
```


ptdis: probability variable from "t" (real) df degrees freedom less than quant

```
real function ptdis(quant,df)
real quant,df
```

ptds: put data structure

```
subroutine ptds(entry, fd)
POINTER entry; integer fd
```

putds: write a dataset

```
subroutine putds(name,entry,pos)
POINTER entry,name; integer pos
```

qantle: quantiles of an ordered sample

```
# note returned quantiles are constrained to be at data points or midway between
subroutine qantle(x,nx,quant,prob,nqp)
integer nx,nqp
real x(nx),quant(nqp),prob(nqp)
```

qbeta: quantiles from the general beta distribution

```
function qbeta(p,a,b)
real p,a,b
# p: probability level
# a,b: parameters
```

qcauc: quartile of cauchy distribution at probability pr

```
real function qcauc(pr,qlocat,qscale)
real pr,qlocat,qscale
```

qf: quantiles from the F distribution

```
real function qf(p,df1,df2)
real p,df1,df2
#return the quantile corresponding to probability p, degrees of freedom
#df1, df2 for numerator, denominator
```

qgamm: quantile of gamma with shape eta at probability p

```
real function qgamm(p,eta)
real p,eta
```

qlnor: quantile of log normal distribution at probability pr

```
real function qlnor(pr,scale,shape)
real pr,scale,shape,qnorms
```

qlogi: quantile of logistic distribution

```
real function qlogi(prob,aloc,scale)
real prob,aloc,scale
```

qnorm: normal family quantile function

```
real function qnorm(p,a,b)
real p,a,b
real qnorms
```

qnorms: quantile of standard normal at probability pr

```
real function qnorms(pr)
real pr
real p,eta,term,f1,f2,f3,f4,f5,f6
```


qtdis: quantile of t distribution df degrees of freedom at probability p

```
real function qtdis(p,df)
real p,df
```

r9lgmc: log gamma correction factor for x greater/equal 10.

```
real function r9lgmc(x)
real x
# august 1977 edition. w. fullerton, c3, los alamos scientific lab.
# compute the log gamma correction factor for x .ge. 10.0 so that
#  $\text{alog}(\text{gamma}(x)) = \text{alog}(\sqrt{2\pi}) + (x-.5)*\text{alog}(x) - x + \text{r9lgmc}(x)$ 
```

rangec: accumulate min and max

```
subroutine rangec(x,n,xl,xu)
integer n; real x(n),xl,xu
```

rangem: min and max of matrix

```
subroutine rangem(a,n,nd,m,al,au)
integer n,nd,m; real a(nd,m),al,au
```

rangev: find min and max of vector

```
subroutine rangev(x,n,xl,xu)
integer n; real x(n),xl,xu
```

rbiwt: biweight regression of y vs single x variable

```
subroutine rbiwt(x,y,n,a,b,sd,wts,ai,bi,xk,maxit,tol)
integer n,maxit
real x(n),y(n),wts(n),a,b,sd,ai,bi,xk,tol
# compute the biweight regression estimates of a and b
# in the model  $y = a + b * x + \text{error}$ .
# sd is the estimated asymptotic residual standard error.
# wts is a scratch vector of length n; returns the final weights.
```

rcopy: copy real data

```
subroutine rcopy(from,to,length)
real from(1),to(1)
integer length,i
```

rdtfmt: find format for real number

```
subroutine rdtfmt(x,n,length,nd,type)
real x(1); integer n,length,nd,type
```

regsum: regression summary

```
subroutine regsum(n,p,r,c,tbl,cov,cor,res,rms,rsq,fval,int)
integer n,p; real r(p,p),c(p),tbl(p,3),cov(p,p),cor(p,p),res(n),rsq,fval,rms
logical int
```

refil: fill TO vector with (repeated) values from FROM vector

```
subroutine refil(from,nfrom,to,nto,mode)
POINTER from,to
integer nfrom,nto,mode
```

rever: reverse entries in a vector

```
subroutine rever(x,n)
real x(n)
```

rexpos: random exponentials

```
real function rexpos(iseed)
integer iseed
real runifs
```


rfill: fill vector of reals

```
subroutine rfill(value,x,n)
integer n,i
real value,x(n)
```

rgamm: random gamma with shape eta

```
real function rgamm(eta,iseed)
real eta; integer iseed
# a pseudo-random gamma generator as presented by
# j.w.frane and v.k. murthy in jasa (to appear in 1974)
# the parameter beta is assumed to be 1.0 .
```

rnormk: Random Normal (Kinderman & Monahan, Proc. ASA Comp., 1978)

```
real function rnormk(iseed)
# calls uni - portable random uniform generator
integer iseed
```

rpois: random poisson with parameter xlam

```
integer function rpois(xlam,iseed)
real xlam
integer iseed
```

rstab: random stable standardized form

```
function rstab(alpha,bprime,u,w)
# arguments ..
# alpha : characteristic exponent
# bprime : skewness in revised parameterization
# u : uniform variate on (0.,1.), for example from a uniform
# pseudo-random number generator
# w : exponentially distributed variate
```

sasum: sum of absolute values of x

```
real function sasum(n,x,incx)
# this function returns the sum of the absolute values
# of the components of x. only every incxth component
# of x is considered
integer n,incx; real x(incx,1)
integer i
```

sattac: attach file for use in S and put onto file list

```
integer function sattac(name,perm,status)
integer perm,status
CHARACTER( name,*)
```

saxpy: multiply x by a and add to y

```
subroutine saxpy(n,a,x,incx,y,incy)
# this subroutine multiplies the x vector by a and
# adds the result to the y vector
# only every incxth component of x and every incy
# component of y are considered
integer n,incx,incy
real x(incx,1),y(incy,1),a
```

sdetac: detach file given file code

```
integer function sdetac(fd)
integer fd
```

search: search one of the lists (chapters, data bases, macros)

```
subroutine search(name,which,path,perm,statck)
POINTER name,path; integer which,perm
```


setprm: print prompt string if terminal (without newline)

```
subroutine setprm(infd, string)
integer infd; CHARACTER(string,*)
```

sopen: open a file for S

```
integer function sopen(name, perm)
CHARACTER(name,*); integer perm
```

sort: sort real array a of length n

```
subroutine sort(a,n)
integer n; real a(n),tt,t
```

sorti: sort integer array a of length n

```
subroutine sorti(a,n)
integer a(n),n,tt,t
```

sortp: sort real array a of length n with passive list

```
subroutine sortp(a,n,b)
integer n; real a(n),tt,t
integer b(n),itt,it
```

sortpi: sort integer array a of length n with passive list

```
subroutine sortpi(a,n,b)
integer n; integer a(n),tt,t
integer b(n),itt,it
```

sortpr: sort real array a of length n with real passive list

```
subroutine sortpr(a,n,b)
integer n; real a(n),tt,t
real b(n),itt,it
```

split: split plateaus using end value rule

```
subroutine split(y,scr,iscr,n)
integer n,iscr(n)
real y(n),scr(n)
```

splitr: x3r with splitting

```
subroutine splitr(x,scr,scr1,scr2,m)
integer m
real x(m),scr(m),scr1(m),scr2(m)
```

sscal: scale every incxth component of x by a

```
subroutine sscal(n,a,x,incx)
integer n,incx
real a,x(incx,n)
```

stem: produce stem and leaf display of x length n

```
subroutine stem(x,n)
real x(n)
integer n
```

stemi: produce stem and leaf display of ix length n

```
subroutine stemi(ix,n)
integer ix(n),n
integer i
```


stems: set up for stemw

```

subroutine stems(a,m,fc,smax1,width1,nl1,fac,same,twodig,nohead,break,fence,integr)
# sets up the options for stemw
integer m,nl1,fc,smax1,width1,width,break
real a(m),fac,fence
logical same,twodig,integr,nohead

```

stemw: produce stem and leaf portion for stem

```

subroutine stemw(a,n,fc,kkl,kku,ks1,nl,wd,twodig)
real a(n)
integer n,fc,kkl,kku,ks1,nl,width,wd
logical twodig

```

streq: compare two strings

```

logical function streq(st1,st2)
CHARACTER( st1,*)
CHARACTER(st2,*)

```

sum: sum of vector x of length n

```

real function sum(x,n)
integer n; real x(n)

```

sumxy: inner product of vectors

```

real function sumxy(x,y,n)
integer n; real x(n),y(n)

```

svd: singular value decomposition

```

subroutine svd(a,m,n,u,nu,s,v,nv)
real a(m,1),u(m,1),v(n,1),s(1)
integer m,n,nu,nv
#a: m by n matrix to decompose
#u: matrix for left factor; returns nu columns nu must be 0,n,m
#s: vector for singular values
#v: matrix for left factor; returns nv columns, nv must be 0 or n

```

tinvit: inverse iteration technique

```

subroutine tinvit(nm,n,d,e,e2,m,w,ind,z,ierr,rv1,rv2,rv3,rv4,rv6)
integer i,j,m,n,p,q,r,s,ii,ip,jj,nm,its,tag,ierr,group
real d(n),e(n),e2(n),w(m),z(nm,m),
      rv1(n),rv2(n),rv3(n),rv4(n),rv6(n)
integer ind(m)
#   this subroutine is a translation of the inverse iteration tech-
#   nique in the algol procedure tristurm by peters and wilkinson.
#   handbook for auto. comp., vol.ii-linear algebra, 418-439(1971).

```

tolowr: convert full matrix to lower triangle

```

# y may overwrite x if desired
subroutine tolowr(x,n,p,y,diag)
integer n,p
real x(n,p),y(1)
logical diag

```

transi: transpose integer matrix in place

```

subroutine transi(a,m,n)
integer a(1)
integer m,n,i,iok

```

transr: transpose real matrix in place

```

subroutine transr(a,m,n)
real a(1)
integer m,n,i,iok

```

transv: transpose data structure

```

subroutine transv(value,mode,nrow,ncol)
POINTER value; integer mode,nrow,ncol

```


trbak1: back transform for eigenvectors

```

subroutine trbak1(nm,n,a,e,m,z)
real a(nm,n),e(n),z(nm,m)
#   this subroutine is a translation of the algol procedure trbak1,
#   num. math. 11, 181-195(1968) by martin, reinsch, and wilkinson.
#   handbook for auto. comp., vol.ii-linear algebra, 212-226(1971).

```

tree: tree from bookkeeping of merging

```

subroutine tree(left,right,height,n,leaves,work)

```

treecn: tree canonicalization from graph

```

subroutine treecn(left,right,height,n)
#   "left"/"right" arguments correspond to the arguments
#   "old"/"new" of related subroutines
real height(1), lh, rh

```

treelv: tree leaves from graph

```

subroutine treelv(old,new,n,list,stack)
#   "old" is on left of graph, "new" is on right
integer old(1), new(1), list(1), stack(1)

```

tridib: eigenvalues by bisection

```

subroutine tridib(n,epsl,d,e,e2,lb,ub,m11,m,w,ind,ierr,rv4,rv5)
integer i,j,k,l,m,n,p,q,r,s,ii,m1,m2,m11,m22,tag,ierr,isturm
real d(n),e(n),e2(n),w(m),rv4(n),rv5(n)
#   this subroutine is a translation of the algol procedure bisect,
#   num. math. 9, 386-393(1967) by barth, martin, and wilkinson.
#   handbook for auto. comp., vol.ii-linear algebra, 249-256(1971).

```

trmean: trimmed mean or median

```
real function trmean(x,n,alpha)
integer n
real x(n),alpha
```

trmnu: trimmed mean from unsorted data

```
real function trmnu(x,n,trim)
integer n
real x(n),trim
```

trvar: trimmed variance of sorted data

```
real function trvar(x,n,alpha)
integer n
real x(n),alpha
```

trvaru: trimmed variance of unsorted data

```
real function trvaru(x,n,trim)
integer n
real x(n),trim
```

twice: apply smoother to series and residuals

```
subroutine twice(x,sx,n,smooth)
integer n
real x(n),sx(n)
external smooth
```

uni: Uniform Random No. Generator

```
real function uni(k)
integer k,ibyte(4)
real rlunif
# uni is returned as a single real random variate
# from the uniform distribution  $0.0 \leq uni < 1.0$ .
```


unlink: remove a file

```
subroutine unlink(name)  
CHARACTER(name,*)
```

uscale: rescale vector to (0,1)

```
subroutine uscale(x,n,xs)  
real x(1),xs(1)
```

jstkgt: allocate dynamic array

```
POINTER function jstkgt(nitems,itype)  
integer nitems,itype
```

var: sample variance of vector

```
real function var(x,n)  
integer n; real x(n)
```

varchr: convert characters (packed) to stack

```
POINTER function varchr(chars,leng)  
CHARACTER(chars,*)  
integer leng  
# allocates stack space for chars without trailing blanks  
# if leng is not positive, searches for terminator
```

vec2ch: pointers to character matrix

```
subroutine vec2ch(p,n,leng,chars,lenout)  
# converts vector of character strings on stack to matrix of characters  
POINTER p(n)  
integer n,leng,lenout  
CHARACTER(chars,*)
```

vhash: hash table search for pointer to int/real/char

```
subroutine vhash(item,mode,tab,tlen,enter,i,status)
integer mode,tlen,i,status; POINTER item,tab(tlen); logical enter
# item: stack pointer to data value
# mode: mode of item
# tab: table of pointer items ( NULL if empty)
# tlen: length of table
# enter: should the item be entered if not found
# i: returned as the index of item  $0 < i \leq tlen$ 
# status: code (NEW_ITEM, OLD_ITEM, TABLE_FULL)
```

x2: medians of length 2

```
subroutine x2(x,sx,n)
integer n; real x(n),sx(n)
```

x3: running medians of 3 with end value rule

```
subroutine x3(y,scr,n)
integer n
real y(n),scr(n)
```

x3r: 3r with end value rule on last iteration

```
subroutine x3r(y,scr,n)
integer n; real y(n),scr(n)
```

x4: running medians of length 4

```
subroutine x4(x,sx,n)
integer n; real x(n),sx(n),y(4)
```


x43s2h: 4(3rsr)2h smoother

subroutine x43s2h(x,sx,n)
integer n; real x(n),sx(n)

xmed3: median of 3 numbers

real function xmed3(a,b,c)
real a,b,c

xmed4: median of 4 numbers

real function xmed4(x)
real x(4)

xmedn: median of sorted data

real function xmedn(x,n)
integer n; real x(n)

xmedu: median of unsorted data

real function xmedu(x,n)
integer n; real x(n)

xor: exclusive or**yenter: put new entry in directory**

POINTER function yenter(str,i,entry)
POINTER str,entry,dirptr,alcdir,pnew; integer i,n,l,lnew,j,ij

zrange: find (cumulative) min and max

```
subroutine zrange(x,n,xl,xu,new)
integer n; real x(n),xl,xu; logical new
```

zrrepz: representation of real number

```
subroutine zrrepz(x,nleft,nright,nexp)
real x; integer nleft,nright,nexp
# returns nleft,nright = no. of digits to left & right of . to represent x
#      nexp = no. of digits in mantissa for exponential representation
```


Graphical Algorithm Detailed Documentation

arfigz: aspect ratio figure

subroutine arfigz(ar)

armfgz: multiple figures with given aspect ratio

subroutine armfgz(i,j,m,n,aratio)

arowsz: plot set of arrows

subroutine arowsz(x1,y1,x2,y2,n)

arpltz: generates a plotting surface of given ratio of width

subroutine arpltz(aratio)

barz: create bar plot and label vertical axis

subroutine barz(h,ndim,ndiv,nbar,w,insid,spaces,nspace,xlusr)

integer ndim,ndiv,nbar,nspace

real h(ndim,nbar),w(nbar),spaces(nspace),xlusr

logical insid

bboxz: multi struck box

subroutine bboxz(n)

beginz: advance to new figure

subroutine beginz

bhtchz: puts hatch marks on divided bars

```
subroutine bhtchz(h,ndim,ndiv,nbar,w,alpha,nshade,pin,color,ncol,spaces,nspace,xlusr)
integer ndim,ndiv,nbar,nspace,ncol
real h(ndim,1),alpha(1),w(1),pin(1),color(ncol),spaces(nspace),xlusr
```

bmglgz: margin legend for bar plots

```
subroutine bmglgz(ndiv,alpha,pin,labls)
real alpha(ndiv),pin(ndiv)
POINTER labls(ndiv)
```

bnamez: labels bars with text

```
subroutine bnamez(h,ndim,ndiv,nbar,w,names,xlusr)
real h(ndim,nbar),w(nbar)
POINTER names(nbar)
```

boxz: draw a box around the plotting region

```
subroutine boxz
```

bplotz: plot mixtures

```
subroutine bplotz(x,y,z,n,title,xlab,ylab,zlab)
CHARACTER(title,*); CHARACTER(xlab,*)
CHARACTER(ylab,*); CHARACTER(zlab,*)
```

brpltz: plots a vector of bars

```
subroutine brpltz(h,n)
real h(1,n),w(1)
```

bxintz: initialize for box plot

```
subroutine bxintz(v,n,nrv)
real v(nrv,n)
```

bxlabz: x axis tick label for box plot

```
subroutine bxlabz(labs,num,gen,widths)
POINTER labs(num)
real widths(num),laline
logical gen,fixed,xpd,same
```

bxnz: draw notched box

```
subroutine bxnz(v,center,halfw,hcap,dpth,dash)
real v(7)
logical dash
```

bxylgz: plot region legend for bar plots

```
subroutine bxylgz(ndiv,alpha,pin,labls,xmn,xmx,ymn,ymx)
real alpha(ndiv),pin(ndiv)
POINTER labls(ndiv)
```

bxz: draw fixed width box

```
subroutine bxz(v,center,halfw,hcap)
real v(5)
```

cirarz: draw circular arc (not an ellipse)

```
subroutine cirarz(x,y,r,ang1,ang2)
```


contrz: contour plot from array of function values

```
subroutine contrz(x,nx,y,ny,ans,nd,value,nk)
real ans(nd,ny),x(nx),y(ny)
real value(nk)
```

crclsz: draw n circles

```
subroutine crclsz(x,y,r,n)
real x(n),y(n),r(n)
```

defltz: set default values for all parameters

```
subroutine defltz
```

diraxz: directly specify axis

```
subroutine diraxz(side,type,p1,p2,p3,umin,umax)
integer side
CHARACTER(type,1)
```

dlinez: lines that miss circular areas around points

```
subroutine dlinez(x,y,n)
real x(n),y(n)
```

dmarkz: draw plotting symbol marks at points

```
subroutine dmarkz(xx,yy,n,marker)
real xx(n),yy(n),x(9),y(9)
```

drfigz: directly specify figure

```
subroutine drfigz(h1,h2,v1,v2)
```

.

drpltz: plot by direct specification

```
subroutine drpltz(pw1,pw2,ph1,ph2)
```

escalz: set up equal scales on x and y axes

```
subroutine escalz(inside,uxmin,uxmax,uymn,uymax,ax,bx,ay,by)
```

faxisz: plot axis with functional transf. (table notation)

```
subroutine faxisz(inside,table,n,fun,axis,labels)
```

```
real table(n)
```

```
external fun
```

```
real fun
```

```
logical axis,labels
```

finisz: finished with plotting

```
subroutine finisz
```

framez: advance to new frame (unconditional)

```
subroutine framez
```

gridz: plot grid and optional subgrid

```
subroutine gridz(nxgr,nygr,subgr,ndx,ndy)
```

```
logical subgr,xpd
```


gtextz: generate text in specified margin position

```
subroutine gtextz(inside,xline,coord,string)
CHARACTER(string,*)
```

hatch: shade in polygon

```
subroutine hatch(x,y,n,space,angle)
integer n; real x(n),y(n),space,angle
```

hhbrkz: create breakpoints for histogram

```
subroutine hhbrkz(z,n,k,nbreak,cbreak)
real z(n),cbreak(*),adjust(2)
```

hhcntz: count the number of observations in each class

```
# if an obs. falls on a class break, it goes to class on right
# assumes sorted break points
subroutine hhcntz(z,n,nbreak,cbreak,class,idens)
real cbreak(nbreak),z(n),class(*)
logical idens
```

hhdonz: number of classes for histogram by Doanes rule

```
subroutine hhdonz(z,n,k,ke)
real z(n)
```

hhistz: histogram

```
subroutine hhistz(z,n,idens,ipcmt,nbrk,cbreak)
real z(n),cbreak(*)
```

hhpltz: histogram

```
subroutine hhpltz(z,n,nb)
real z(n),cbreak(1)
```

inmfgz: multiple figures specified by inches

```
subroutine inmfgz(i,j,m,n,w,h)
```

inpltz: plotting surface specified in inches

```
subroutine inpltz(width,height)
```

inputz: read vector of (x,y) points from device

```
subroutine inputz(x,y,n,nmax)
integer nmax,n
real x(1),y(1)
```

intz: plot integer at (x,y)

```
subroutine intz(x,y,i)
real x,y; integer i
```

laxisz: plot logarithmic axis

```
subroutine laxisz(iside,axis,labels)
logical axis,labels
```

lclipz: clip a line segment to given bounds

```
subroutine lclipz(xi,yi,bx,by,x,y,jint)
real x(2),y(2),bx(2),by(2),xi(2),yi(2)
```


legnhz: plot text string with character count

```
subroutine legnhz(x,y,it,nch,pos)
CHARACTER(it,*)
real x,y,pos
integer nch
```

legnpz: plot text string

```
subroutine legnpz(x,y,string,pos)
CHARACTER(string,*)
```

lenstz: length in inches and character count

```
subroutine lenstz(it,length,ncact)
```

linesp: lines with picture transformation

```
subroutine linesp(x,y,n)
real x(n),y(n)
```

linesz: connected line segments

```
subroutine linesz(x,y,n)
real x(n),y(n)
```

lintfz: plot lines for transformed data

```
subroutine lintfz(x,y,n,ixf,xfun,yfun)
real x(n),y(n),xx(10),yy(10)
external xfun,yfun
real xfun,yfun
```

logaxz: determine and plot log axis

subroutine logaxz(inside,istyle,u1,u2,nl,isw)

lrngez: setup for log axes

subroutine lrngez(x,y,n,ixy)
real x(n),y(n)

lspltz: scatter plot with logarithmic axes

subroutine lspltz(x,y,n,ixy)
real x(n),y(n)

mfcolz: multiple figures per page by columns

subroutine mfcolz(m,n)

mfrowz: multiple figures per page by rows

subroutine mfrowz(m,n)

mintvz: multiple pretty intervals with same range

subroutine mintvz(xi,xu,n,ninti,xo,ri,dr,nint)
real xi(n),ri(n),xo(n),xu(n)

mtextz: general routine for text in margin

subroutine mtextz(inside,xline,string)
CHARACTER(string,*)

mxfigz: maximum sized figure

```
subroutine mxfigz
call mxmfgz(1,1,1,1)
return
end
```

mxmfgz: maximum size multiple figures on page

```
subroutine mxmfgz(i,j,m,n)
```

mxpltz: generates maximum plot surface allowing for margins

```
subroutine mxpltz
```

nextz: advance to next figure

```
subroutine nextz
```

nplotz: scatter plot with repeated points

```
subroutine nplotz(x,y,n)
real x(n),y(n)
```

npntsz: plot points reporting multiples

```
subroutine npntsz(x,y,n)
real x(n),y(n)
```

otextz: text in outer margin

```
subroutine otextz(side,line,text)
CHARACTER(text,*)
integer side
real line
```

pdumpz: dump some or all of parameters

```
subroutine pdumpz(file,which)
integer file; logical which(1)
```

pictur: picture setup (transforms) before linesp

```
subroutine pictur(xp,yp,e,r)
```

piez: pie chart

```
subroutine piez(x,n,label,frac,inner,outer)
real x(n),frac,inner,outer
POINTER label(n)
```

plarcz: plot circular or elliptical arcs

```
subroutine plarcz(xc,yc,r,a1,a2)
```

pltusa: plot map of United States

```
subroutine pltusa
```

pnttfz: plot points for transformed data

```
subroutine pnttfz(x,y,n,ixf,xfun,yfun)
real x(n),y(n),xx(10),yy(10)
external xfun,yfun
real xfun,yfun
```

pointz: plot set of points

```
subroutine pointz(x,y,n)
real x(n),y(n)
```


ptitle: title a plot

```
subroutine ptitle(ttl,sub,xlab,ylab)
CHARACTER(ttl,*); CHARACTER(sub,*); CHARACTER(xlab,*); CHARACTER(ylab,*)
```

realz: plot real at (x,y)

```
subroutine realz(x,y,r)
real x,y,r
```

robaxz: robust definition of standard axis

```
subroutine robaxz(iside,istyle,u,n,nint,rbd,bd,obd,isw)
```

rplotz: robust scatter plot

```
subroutine rplotz(x,y,n)
real x(n),y(n)
```

rpntsz: robust plot of scatter of points

```
subroutine rpntsz(x,y,n,rbd,bd,obd)
real x(n),y(n),rbd(4),bd(4),obd(4)
```

saxisz: draw a standard axis

```
subroutine saxisz(iside,axis,labels)
logical axis,labels
```

segmtz: set of disconnected line segments

```
subroutine segmtz(x1,y1,x2,y2,n)
real x1(n),x2(n),y1(n),y2(n)
```

shadez: shades in rectangle

subroutine shadez(alpha,pin,x1,x2,y1,y2,xref,yref)

sleep: suspend execution for time seconds

subroutine sleep(time)

integer time

splotz: general purpose scatter plot routine

subroutine splotz(x,y,n)

real x(n),y(n)

sqfigz: square figure

subroutine sqfigz

sqmfgz: multiple figures each with square figure space

subroutine sqmfgz(i,j,m,n)

sqpltz: set up square plot

subroutine sqpltz

srngez: set up scatter plot

subroutine srngez(x,y,n)

real x(n),y(n)

stdaxz: standard (linear) axis

subroutine stdaxz(iside,istyle,u1,u2,nin,isw)

taxisz: generate time axis from stored parameters

subroutine taxisz(iside,axis,labels)
logical axis,labels,yaxis

textcz: centered text

subroutine textcz(x,y,text)
CHARACTER(text,*)

textz: plot text string

subroutine textz(x,y,text)

timaxz: specify time axis

subroutine timaxz(iside,istyle,u1,u2,period,isw)

titlez: plot title

subroutine titlez(title,xlab,ylab)
CHARACTER(title,*); CHARACTER(xlab,*); CHARACTER(ylab,*)

trngez: set up ranges etc for time series plot

subroutine trngez(y,start,end,period)
real y(1)

tsaxpz: plot axis in time format

subroutine tsaxpz(iside,xllab,xlline,y1,y2,nint,pa,leng,iswa)

tsvecz: set up vector for time series

subroutine tsvecz(x,start,end,period)
real x(1)

uaxisz: plot linear axis at specified user coordinates

subroutine uaxisz(iside,u,axis,labels)
logical axis,labels

usabdy: coordinates of USA boundary

subroutine usabdy(xout,yout,n,iseg)
real xout(1),yout(1)

xtickz: vertical ticks

subroutine xtickz(vec,n,begin,end)
real vec(n),begin,end

ytickz: horizontal ticks

subroutine ytickz(vec,n,begin,end)
real vec(n),begin,end

zarowz: arrow drawing

subroutine zarowz(x1,y1,x2,y2,w,open,rel)

zoutrz: send ascii characters to terminal

Index

Index to S User's Guide

- [, subset operator 2-2
- ^, exponentiation operator 2-1
- { compound expression 3-6
- |, or operator 2-1
- !, not operator 2-1
 - operating system escape 2-4
- !=, not equal operator 2-1
- #, comment 2-3, 5-2
- \$ dataset name (without prefix) 3-12
- % LOOP macro argument 4-8
- * macro argument 4-7, 4-13
- \$, component selection operator 2-11
- \$T intermediate dataset 3-3, 4-2
- %%, modulo operator 2-1
 - remainder operator 2-1
- %* operator 3-22
- %, special operator 5-1
- %/ , integer division operator 2-1
- %c operator 3-22
- %m, coerce operator 3-7
- %o operator 3-22
- & CHAIN continuation 6-22
 - ENDARGS suppressing 6-18, 8-11
- &, and operator 2-1
- ({ macro compound expression 3-3, 4-4
- **, exponentiation operator 2-1
- *, multiplication operator 2-1
- + continuation prompt 2-3
- +, addition operator 2-1
- , subtraction operator 2-1
- /, division operator 2-1
- :, sequence operator 2-1
- <, less than operator 2-1
- <-, assignment operator 2-1, 2-3
- <=, less than or equal operator 2-1
- =, named argument 2-3
- ==, equal operator 2-1, 2-3
- > prompt 2-1
- >, greater than operator 2-1
- >=, greater than or equal operator 2-1
- ? macro invocation 3-1, 4-1, 5-1
- ?(macro literal 4-5
- abbreviation, argument name 2-3
 - component name 2-11
- abline function 3-36
- ABORT statement 7-2, 7-5
- acos function 2-15
- actual argument 6-17
- addition operator + 2-1
- adj graphical parameter 3-32, 3-36, 8-7, 8-18
- again function 2-4
- algebra algorithms, linear 7-18
- algorithm facility 7-4
 - facility, attach 7-11
 - graphics 8-4
 - io 7-6
 - print 7-4
 - read 7-8
 - stack 7-11
 - struct 7-12
 - tree 7-13
- language 6-1, 7-1, 8-4
- algorithm,
 - cfill 7-14
 - ch2vec 7-15
 - chcomp 7-16
 - concat 7-14
 - icopy 7-14
 - ifill 7-14
 - istrng 7-6, 7-15
 - jgetch 7-15
 - jputch 7-15
 - jstkgf 7-11
 - jstkrl 7-12
 - lcopy 7-14
 - lfill 7-14
 - match 6-11
 - narang 7-17, 8-5
 - orderf 7-16
 - orderi 7-16
 - orderr 7-16
 - orderv 7-16
 - rangec 7-16, 8-5
 - rangev 7-16, 8-5
 - rcopy 7-14
 - rdtfmt 7-7
 - rfill 7-14
 - sattac 7-11
 - sclose 7-11
 - sdetac 7-11
 - sopen 7-11
 - sort 7-15
 - sortfr 7-15
 - sorti 7-15
 - sortv 7-16
 - streq 7-15
- algorithms, graphical 8-1
 - linear algebra 7-18
 - probability 7-17
 - quantile 7-17

random number 7-17
 all function 2-15
 ALLARG statement 6-19
 allocation, dynamic storage 7-11
 scratch space 6-4
 alphanumeric mode 8-28
 analysis, multivariate 3-20
 and operator & 2-1
 any function 2-15
 ANY specifier 6-14
 aperm function 3-11
 append function 2-6
 apply function 3-3
 approx function 3-38
 arbitrary number of macro argument 4-7
 ARG statement 6-18
 ARGSTR statement 6-21
 argument 5-5
 =, named 2-3
 default value, macro 3-2, 4-3
 list from structure 6-20
 name abbreviation 2-3
 partial match 5-6, 6-17
 processing 5-6
 structure 5-5
 argument, \$% LOOP macro 4-8
 \$* macro 4-7
 actual 6-17
 arbitrary number of macro 4-7
 FILTER 6-22
 formal 6-17
 function 2-2
 macro named 4-7
 missing 5-6
 MISSING 6-3
 mode of 6-2
 named 2-2, 6-18
 optional 2-2
 OPTIONAL 6-3
 optional 6-3
 PAR 6-22, 6-24
 plot= 3-29
 PLOTARGS 6-25
 positional macro 4-3
 type of 6-2
 USED 6-21
 white space in macro 4-6
 arithmetic operator 2-1
 arithmetic, operands unequal length 3-7
 time-series 2-10
 arowsz graphical subroutine 8-5
 arpltz graphical subroutine 8-17
 array 3-11

function 3-11
 array, re-arranging 3-11
 arrow (see assignment) 2-1
 asin function 2-15
 assign function 3-10, 5-7, 5-10
 assigning datasets 3-10
 assignment 2-1
 operator <- 2-1, 2-3
 assignment, with prefix 3-12
 atan function 2-15
 attach algorithm facility 7-11
 database 3-13
 function 3-13
 attribute 6-5
 attribute, DATANAME 6-12
 ENTRY 7-13
 FIRSTENT 7-13
 FNAME 6-12
 KEYNAME 6-12
 LASTENT 7-13
 LENGTH 6-5, 6-10, 7-12
 MISSING 6-5, 6-17
 MODE 6-10, 7-12
 NAME 7-12
 NCOL 6-5, 6-10
 NROW 6-5, 6-10
 STRING 6-11
 TEND 6-5, 6-10
 TEXT 6-11
 TNPER 6-5, 6-10
 TSTART 6-5, 6-10
 VALUE 6-17, 7-12
 axes function 6-24, 8-12
 axis labels (pretty) 8-21
 labels, numerical values for 3-33
 numeric labels 8-6
 parameters 8-22
 AXIS statement 8-5, 8-20
 axis style 8-21, 8-22
 tick marks 8-6
 type 8-22
 axis, logarithmic 8-21
 time 8-21
 backsolve function 3-22
 bar plot 3-25
 barplot function 3-25, 3-29
 barz graphical subroutine 8-25
 batch execution 3-9
 BATCH utility 3-9
 beginz graphical subroutine 8-4
 beta distribution 2-12
 bhtchz graphical subroutine 8-25
 BIG constant 7-3
 BIGEXP constant 7-3

- bivariate interpolation 3-27
- bmglgz graphical subroutine 8-25
- bnamez graphical subroutine 8-25
- boolean operator 2-1
- BOTTOM side of plot 8-6
- box function 3-38
 - plot 3-23
- boxplot function 3-23, 3-29
- BPLOT utility 3-39
- bplotz graphical subroutine 8-25
- break syntax 5-9
- buffer size, macro 4-11
- BUFFER variable 7-6
- BUFLen variable 7-6
- BUFPOS variable 7-6
- built-in dataset 3-6, 5-9
 - mode 6-16
 - modes 3-7
 - NULL 3-16
- bxnz graphical subroutine 8-25
- bxp function 3-24
- bxpz graphical subroutine 8-25
- bxz graphical subroutine 8-25
- C format item 7-5, 7-9
- c function 2-13, 3-5, 3-23
- C language 7-1, 7-4
- cancor function 3-20
- canonical correlation 3-20
- case sensitive language 1-2
- CASE statement 6-15
- category 3-16
- Cauchy distribution 2-12
- cbind function 2-9
- ceiling function 2-16
- cex graphical parameter 3-31, 8-8
- cfill algorithm 7-14
- ch2vec algorithm 7-15
- CHAIN continuation, & 6-22
 - statement 6-23, 8-12
- changing graphical parameters 8-3
- chapter function 5-5, 6-7
- CHAPTER utility 6-6
- CHAR 3-7
 - mode 7-3
- CHARACTER statement 7-3
- character string 2-5, 5-1
 - data 6-10, 7-6, 7-14
 - delimiter 2-5
 - string, reading 2-6
- character, EOS 6-11, 7-15
- chcomp algorithm 7-16
- chi-square distribution 2-12
- chol function 3-22
- Choleski decomposition 7-18
- CLEAR statement 7-7
- clipping 8-19
- closing, file 7-11
- cluster tree plot 3-29
- clustering, hierarchical 3-21
- code function 3-16
- coerce 3-7, 6-3
 - function 3-7
 - operator %m 3-7
- COERCE statement 6-16
- col function 2-9
 - graphical parameter 2-19, 3-32, 8-8
 - macro 3-4
- COMMA format item 7-8
- command 5-3
- command, pwd 6-7
- comment # 2-3, 5-2
- common block declaration 6-5
- compilation 6-7
- compname function 3-23
- component insertion, data structure 6-23
 - name abbreviation 2-11
 - select 6-13
 - selection operator \$ 2-11
- component, Data 3-12
 - data structure 2-11, 3-5, 3-8
 - Dim 3-12
 - Tsp 3-12
- components of datasets 3-10
- compound expression 3-3, 3-5, 3-6, 4-3, 5-4, 5-8
 - expression, value of 3-7, 5-8
 - { 3-6
- concat algorithm 7-14
- conditional expression 3-6, 5-8
 - expression, value of 3-6
- consistency of graphical parameters 8-18
- constant, BIG 7-3
 - BIGEXP 7-3
 - DEG2RD 7-3
 - EOS 7-3, 7-7
 - ESCAPE 7-3
 - FIELDWIDTH 7-6
 - LARGEINT 7-3
 - LINEWIDTH 7-6
 - NBPC 7-3
 - NCPW 7-3
 - NDIGITS 7-3
 - PI 7-3
 - PRECISION 7-3
 - SMALL 7-3
- cont2z graphical subroutine 8-25
- continuation 5-4
 - prompt, + 2-3

continuation, expression 2-3
 contour function 3-26
 plot 3-26
 contrz graphical subroutine 8-25
 coordinate system, device 8-26
 margin 8-2, 8-18
 raster 8-26
 user 3-33, 8-2, 8-20,
 8-22
 cor function 2-12
 correlation scaling 3-20
 correlation, canonical 3-20
 cor function 2-12
 cos function 2-15
 covariance, var function 2-12
 Cp macro 3-19
 plot 3-19
 crclsz graphical subroutine 8-5
 creating labels 3-23
 crt graphical parameter 3-32, 8-7
 csi graphical parameter 8-8
 csr graphical parameter 8-8
 cstr function 3-23
 csweep macro 3-4
 CTABLE statement 6-11
 cumsum function 2-16
 cut function 3-16
 cutree function 3-21
 cxy graphical parameter 8-7
 cycle function 2-10
 D> macro definition prompt 4-2
 Data component 3-12
 data structure 2-8, 5-9, 7-12
 component 2-11, 3-5, 3-8
 insertion 6-23
 structure, depth-first scan of 7-13
 dynamically allocated 6-4
 external representation of
 5-10
 graphical 3-29
 internal representation of
 5-10
 NA in 6-12
 plot 6-25
 self-describing 2-5
 data, character string 6-10, 7-6, 7-14
 discrete-valued 3-16
 missing (see NA) 2-5
 pointer 6-11
 qualitative 3-16
 read from file 2-6
 terminal 2-6
 database 1-1, 2-7
 search list 2-7

database, attach 3-13
 detach 3-13
 list of names 2-7
 removing dataset from 2-7
 save 2-7
 shared 2-7
 system 2-7
 working 2-7
 DATANAME attribute 6-12
 dataset 1-1, 2-6, 2-7
 documentation 3-17
 name (without prefix), \$ 3-12
 state.center 3-24
 dataset, built-in 3-6, 5-9
 intermediate 3-2
 list of names 2-7
 macro 4-1
 Random.seed 7-17
 removing 2-7
 intermediate 4-4
 datasets, assigning 3-10
 components of 3-10
 getting 3-10
 DEBUG statement 6-8, 7-3
 declaration, common block 6-5
 DECODE statement 7-9
 decomposition, Choleski 7-18
 q-r 7-18
 singular-value 7-18
 decompositions, matrix 3-22
 default value, macro argument 3-2, 4-3
 defer function 3-39
 deferred graphics 3-9, 3-38
 graphics, replotting 3-39
 define function 3-2, 4-2
 DEFINE macro 4-11, 4-12
 definition, macro 3-2
 DEG2RD constant 7-3
 delimiter, character string 2-5
 dendrogram plot 3-29
 density function 3-29
 depth-first scan of data structure 7-13
 description list 6-2
 descriptor file 3-15
 descriptor, file 7-11
 detach database 3-13
 function 3-14
 detailed documentation 2-4
 device coordinate system 8-26
 driver, graphical 8-14, 8-26
 stand-alone 8-25
 device, graphical 2-16
 DEVICE_DRIVER statement 8-30
 dget function 3-14, 5-10

- diag function 2-9
- diagonal of matrix 2-9
- diary function 3-9
- diff function 2-15, 2-16
- Dim component 3-12
- din graphical parameter 8-19
- diraxz graphical subroutine 8-22
- discr function 3-21
 - macro 3-21
- discrete-valued data 3-16
- discriminant analysis 3-21
- dist function 3-21
- distance matrix 3-21
- distribution, beta 2-12
 - Cauchy 2-12
 - chi-square 2-12
 - f 2-12
 - gamma 2-12
 - log-normal 2-12
 - logistic 2-12
 - normal 2-12
 - stable 2-12
 - statistical 2-12
 - Student's t 2-12
 - uniform 2-12
- distribution-free statistics 2-14
- distributions, parameters for 2-13
 - probability plots of general 3-26
- division operator / 2-1
- division, integer %/ 2-1
- documentation, dataset 3-17
 - detailed 2-4
 - function 6-7
 - macro 3-17, 4-5
- dot, plotting 2-19
- dput function 3-14, 5-10
- drfigz graphical subroutine 8-18
- driver, graphical device 8-14
- drpltz graphical subroutine 8-17
- dump function 3-14, 5-10
- dumpaz graphical subroutine 8-24
- Dumped expression 2-4
- dynamic storage allocation 7-11
- dynamically allocated data structure 6-4
- edit function 2-4, 3-26
- EDITDOC utility 3-17, 6-8
- editor, macro 3-2
 - text 2-4, 3-1, 3-15, 3-17, 4-2, 4-3
- eepltz graphical subroutine 8-25
- EFL language 7-1
- eigen function 3-22
- eigenvalue 7-18
- else syntax 3-6
- empirical q-q plot 2-18
- encode function 2-9, 3-23
- ENCODE statement 7-6
- end function 2-10
 - of file 7-10
- end, time-series parameter 2-10
- ENDARGS statement 6-18
 - suppressing, & 6-18, 8-11
- ENDARGS, suppression 6-18
- ENTRY attribute 7-13
- environment, S 7-1
- EOF variable 7-10
- EOS character 6-11, 7-15
 - constant 7-3, 7-7
- EPRINT statement 7-5
- eqpltz graphical subroutine 8-25
- equal operator == 2-1, 2-3
- equations, linear 3-22
 - triangular systems of 3-22
- error 2-3
 - message 6-3, 7-2
- error, execution 2-4
 - syntax 2-3
- ESCAPE constant 7-3
- escape to operating system 3-10
- evaluation time 3-3
- execution error 2-4
 - time 4-2
- execution, interrupting 2-4
- exp function 2-15
- expansion, macro 4-1
- exponentiation operator ** 2-1
 - ^ 2-1
- expression 1-1, 2-1, 5-2
 - continuation 2-3
- expression, compound 3-3, 3-5, 3-6, 4-3, 5-4, 5-8
 - conditional 3-6, 5-8
 - Dumped 2-4
 - iterative 5-9
 - value of compound 5-8
- external representation of data structure 5-10
- extract macro 5-10
 - utility 3-15, 5-10, 7-9
- extract, macro 3-15
- f distribution 2-12
- face plot 3-27
- faces function 3-27
- FALSE 2-1, 2-5
- FATAL macro 4-9
 - statement 6-4, 7-2
- FIELD variable 7-10
- FIELDPOS variable 7-10

- FIELDWIDTH constant 7-6
- figure region, graphical 3-33
 - type, graphical 8-18
- figure, graphical 8-2, 8-16
- file 2-6, 5-10
 - closing 7-11
 - descriptor 7-11
 - name suffix 6-6, 7-1
 - opening 7-10
- file, command input from 3-1
 - descriptor 3-15
 - end of 7-10
 - printing sent to 3-1
- FILTER argument 6-22
- fin graphical parameter 8-19
- finisz graphical subroutine 8-8
- FIRSTENT attribute 7-13
- floor function 2-16
- FNAME attribute 6-12
- for loop 3-5
 - syntax 3-5, 5-9
- formal argument 6-17
- format item, C 7-5, 7-9
 - COMMA 7-8
 - I 7-5, 7-9
 - L 7-5, 7-9
 - Q 7-5
 - R 7-5, 7-9
 - S 7-9
 - SHARP 7-8
 - SP 7-7
 - T 7-7
 - TM 7-7
- FORTTRAN language 6-1, 7-1
- FPRINT statement 7-5
- frame function 2-19
- FROM specifier 6-13
- fty graphical parameter 8-18
- function 2-2
 - argument 2-2
 - call 5-5
 - chaining 5-7
 - documentation 6-7
- FUNCTION statement 6-2
 - utility 6-6, 6-24, 7-1, 7-17, 8-10, 8-29
- function, abline 3-36
 - acos 2-15
 - again 2-4
 - all 2-15
 - any 2-15
 - aperm 3-11
 - append 2-6
 - apply 3-3
 - approx 3-38
 - array 3-11
 - asin 2-15
 - assign 3-10, 5-7, 5-10
 - atan 2-15
 - attach 3-13
 - axes 6-24, 8-12
 - backsolve 3-22
 - barplot 3-25, 3-29
 - box 3-38
 - boxplot 3-23, 3-29
 - bxp 3-24
 - c 2-13, 3-5, 3-23
 - cancor 3-20
 - cbind 2-9
 - ceiling 2-16
 - chapter 5-5, 6-7
 - chol 3-22
 - code 3-16
 - coerce 3-7
 - col 2-9
 - compname 3-23
 - contour 3-26
 - cor 2-12
 - cos 2-15
 - cstr 3-23
 - cumsum 2-16
 - cut 3-16
 - cutree 3-21
 - cycle 2-10
 - defer 3-39
 - define 3-2, 4-2
 - density 3-29
 - detach 3-14
 - dget 3-14, 5-10
 - diag 2-9
 - diary 3-9
 - diff 2-15, 2-16
 - discr 3-21
 - dist 3-21
 - dput 3-14, 5-10
 - dump 3-14, 5-10
 - edit 2-4, 3-26
 - eigen 3-22
 - encode 2-9, 3-23
 - end 2-10
 - exp 2-15
 - faces 3-27
 - floor 2-16
 - frame 2-19
 - get 3-10, 5-10
 - gs 3-22
 - hclust 3-21
 - help 2-4, 3-17, 4-5

hist 2-18, 3-29
hp72f 2-16
hp72h 2-16
hp72v 2-16
identify 3-29, 3-30
ifelse 3-6
interp 3-27
l1fit 2-12
labclust 3-29
lag 2-15
leaps 3-18
len 2-6, 2-14
lines 2-19, 3-36
list 2-7, 2-20, 3-13
log 2-15
log10 2-15
loglin 3-19
lowess 3-29
macro 5-2
mail 2-4
match 2-14
matlines 3-37
matplot 2-17, 3-33
matpoints 3-37
matrix 2-8
max 2-16
mdscale 3-22
mean 2-11
median 2-11
medit 3-2, 4-3
min 2-16
mode 2-6
mprint 3-2, 4-3
mstr 3-23
mtext 3-37
NA 2-5
ncomp 3-8, 3-23
nper 2-10
options 3-1, 4-11, 5-8
order 2-14
pairs 3-27
par 3-34
pardump 3-33, 8-24
pbeta 3-20
pcauchy 3-20
pchisq 3-20
persp 3-26
pf 3-20
pgamma 3-20
photo 3-38
pie 3-25
plclust 3-29
plnorm 3-20
plogis 3-20

plot 2-17, 2-21
plotfit 3-25
pmax 2-16
pmin 2-16
pnorm 3-20
points 2-19, 3-36
ppoints 3-38
prcomp 3-20
prefix 3-12
pretty 3-33
print 2-9, 2-20
printer 2-16, 2-21
prism 3-38
prod 2-16
pstab 3-20
pt 3-20
punif 3-20
q 2-23
qbeta 3-20
qcauchy 3-20
qchisq 3-20
qf 3-20
qgamma 3-20
qlnorm 3-20
qlogis 3-20
qnorm 3-20
qqgamma 3-26
qqnorm 2-18, 3-29
qqplot 2-18, 3-29
qstab 3-20
qt 3-20
qunif 3-20
range 2-16, 3-36
rank 2-14
rbeta 2-12
rbind 2-9
rbiwt 2-12
rcauchy 2-12
rchisq 2-12
rdpen 3-29, 3-30
read 2-6, 5-10
reg 3-19
regprt 3-19
regress 2-11, 2-12, 2-22
regsum 3-19
rep 2-13
replace 2-6
replot 3-40
restore 3-14, 5-10
rev 2-14
rf 2-12
rgamma 2-12
rlnorm 2-12
rlogis 2-12

rm 2-7, 3-13
 rnorm 2-12
 round 2-16
 row 2-9
 rreg 3-18
 rstab 2-12
 rt 2-12
 runif 2-12
 sapply 3-4
 save 2-7, 5-7
 scale 3-20
 search 3-14, 6-7
 segments 3-37
 select 3-10
 seq 2-13
 show 2-19, 2-21
 sin 2-15
 sink 3-1
 smatrix 3-28
 smooth 2-15
 solve 3-22
 sort 2-14
 source 3-1
 split 3-5, 3-22
 sqrt 2-15
 stare 3-38
 stars 3-27
 starsymb 3-28
 start 2-10
 stem 2-18, 2-20, 2-23
 subtree 3-21
 sum 2-16
 svd 3-22
 sweep 3-4
 sys 3-10
 t 2-9
 table 3-17, 3-19
 tek10 2-16
 tek12 2-16
 tek14 2-16
 tek14q 2-16
 tek46f 2-16
 tek46h 2-16
 tek46v 2-16
 text 2-19, 3-36
 time 2-10
 title 2-19
 tprint 3-17, 3-19
 trunc 2-16
 ts 2-10
 tslines 3-37
 tsmatrix 2-10
 tsplot 2-17, 3-33
 tspoints 3-37

twoway 2-12
 unique 2-15
 usa 3-24
 value= argument to rm 4-4
 var 2-12
 vu 3-26
 window 2-11
 write 3-14, 5-10
 functions, new 3-1, 6-1
 probability 2-13, 3-20
 quantile 2-13, 3-20
 gamma distribution 2-12
 probability plot 3-26
 general distributions, probability plots of
 3-26
 get function 3-10, 5-10
 GETLINE statement 7-10
 GETSEED statement 7-17
 GETSTRING statement 7-10
 getting datasets 3-10
 graph mode 8-28
 GRAPH utility 8-25
 graphical algorithms 8-1
 data structure 3-29
 device driver 8-14, 8-26
 device, hp72f 2-16
 hp72h 2-16
 hp72v 2-16
 photo 3-38
 printer 2-16
 prism 3-38
 stare 3-38
 tek10 2-16
 tek12 2-16
 tek14 2-16
 tek14q 2-16
 tek46f 2-16
 tek46h 2-16
 tek46v 2-16
 figure 8-2, 8-16
 region 3-33
 type 8-18
 input 3-30, 8-24, 8-29
 margins 3-33, 8-2, 8-16
 outer margins 3-34, 3-37, 8-2,
 8-16
 parameter, adj 3-32, 3-36, 8-7,
 8-18
 cex 3-31, 8-8
 col 2-19, 3-32, 8-8
 crt 3-32, 8-7
 csi 8-8
 csr 8-8
 cxy 8-7

- din 8-19
- fin 8-19
- fty 8-18
- lab 3-32, 8-21
- las 3-31, 8-21
- log 2-17
- lty 2-19, 3-31, 8-8
- lwd 8-8
- mai 8-17
- main 3-33
- mar 3-34, 8-17
- mex 8-17, 8-18
- mfc01 3-34
- mfg 8-17
- mfrow 3-34
- mgp 8-21
- oma 8-17
- omi 8-17
- omo 8-19
- PAR 8-11
- pch 2-19, 3-31
- pin 8-19
- pty 3-34, 8-17
- srt 3-32, 3-36, 8-7
- sub 3-33
- tck 3-32, 8-21
- type 2-17, 3-33
- uin 8-20
- usr 3-35, 8-20, 8-22
- xaxp 8-22
- xaxs 3-32, 8-22
- xaxt 3-32, 8-22
- xlab 3-33
- xlim 3-33
- xpd 8-19
- yaxp 8-22
- yaxs 3-32, 8-22
- yaxt 3-32, 8-22
- ylab 3-33
- ylim 3-33
- parameters 3-31, 8-3
 - PAR 6-22, 6-24
- parameters, changing 8-3
 - consistency of 8-18
 - permanent 3-34
 - temporary 3-31
- plot 8-2, 8-16
 - region 3-33
 - type 8-17
- subroutine, mfcolz 8-17
 - mfrowz 8-17
- subroutine, arowsz 8-5
 - arpltz 8-17
 - barz 8-25
- beginz 8-4
- bhtchz 8-25
- bmglgz 8-25
- bnamez 8-25
- bplotz 8-25
- bxnz 8-25
- bxpz 8-25
- bxz 8-25
- cont2z 8-25
- contrz 8-25
- crclsz 8-5
- diraxz 8-22
- drfigz 8-18
- drpltz 8-17
- dumpaz 8-24
- eepltz 8-25
- eqpltz 8-25
- finisz 8-8
- gtextz 8-19
- hatch 8-25
- hdlinz 8-25
- hhbrkz 8-25
- hhdonz 8-25
- hhpltz 8-25
- i6to9z 8-29
- inpltz 8-17
- inputz 8-24
- laxisz 8-22
- linesp 8-25
- linesz 8-5
- lintfz 8-25
- logaxz 8-21
- mtextz 8-6, 8-18
- nplotz 8-25
- npntsz 8-25
- otextz 8-19
- persp 8-25
- pictur 8-25
- piez 8-25
- plot1 8-25
- pltusa 8-25
- pnrtfz 8-25
- pointz 8-5
- ptitle 8-25
- rplotz 8-25
- rpntsz 8-25
- saxisz 8-6, 8-22
- segmtz 8-5
- shadez 8-25
- splotz 8-25
- stdaxz 8-21
- taxisz 8-22
- textz 8-6
- timaxz 8-21

- titlez 8-25
 - tspltz 8-25
 - tsvecz 8-25
 - usabdy 8-25
 - zejecz 8-28
 - zinitz 8-15
 - zlinez 8-28
 - zoutrz 8-29
 - zoutwz 8-29
 - zparmz 8-27
 - zptchz 8-28
 - zquxyz 8-29
 - zseekz 8-28
 - zwrapz 8-28
- terminal 2-16
- title 8-6
- utility, pf 8-25
 - rdpen 8-26
- graphics algorithm facility 8-4
- graphics, deferred 3-9, 3-38
 - replotting deferred 3-39
- greater than operator > 2-1
 - or equal operator >= 2-1
- gs function 3-22
- gtextz graphical subroutine 8-19
- hatch graphical subroutine 8-25
- hclust function 3-21
- hdlinz graphical subroutine 8-25
- help function 2-4, 3-17, 4-5
- Hewlett-Packard plotter 2-16
- hhbrkz graphical subroutine 8-25
- hhdonz graphical subroutine 8-25
- hhpltz graphical subroutine 8-25
- hierarchical clustering 3-21
 - structure 3-23, 6-13
- hist function 2-18, 3-29
- histogram plot 2-18
- hp72f function 2-16
- hp72h function 2-16
- hp72v function 2-16
- I format item 7-5, 7-9
- i6to9z graphical subroutine 8-29
- icopy algorithm 7-14
- identify function 3-29, 3-30
- identity matrix 2-9
- if syntax 3-6
- IFDEF macro 4-12
- ifelse function 3-6
- IFELSE macro 4-12
- IFELSE, macro 4-6
- ifill algorithm 7-14
- INBUF variable 7-8
- INCLUDE statement 7-4
- INLEN variable 7-8
- inpltz graphical subroutine 8-17
- INPOS variable 7-8
- input, graphical 3-30, 8-24, 8-29
- inputz graphical subroutine 8-24
- INSERT statement 6-23
- insertion, data structure component 6-23
- INT 3-7
 - mode 7-3
- integer division operator %/ 2-1
 - number 2-5
- interactive computing 1-2
- interface language 6-1
 - subset 6-5
 - language, operating on NA in 6-12
- intermediate dataset 3-2
 - dataset, \$T 3-3, 4-2
 - removing 4-4
- internal representation of data structure 5-10
- interp function 3-27
- interpolation, bivariate 3-27
- interpreter 5-1
- interrupting execution 2-4
- io algorithm facility 7-6
- istrng algorithm 7-6, 7-15
- iteration 3-5
- iteration, null value in 3-16
- iterative expression 5-9
- jgetch algorithm 7-15
- jputch algorithm 7-15
- jstkgz algorithm 7-11
- jstkrz algorithm 7-12
- KEYNAME attribute 6-12
- L format item 7-5, 7-9
- llfit function 2-12
- lab graphical parameter 3-32, 8-21
- labclust function 3-29
- labels, creating 3-23
- lag function 2-15
- language, algorithm 6-1, 7-1, 8-4
 - C 7-1, 7-4
 - EFL 7-1
 - FORTRAN 6-1, 7-1
 - interface 6-1
 - RATFOR 6-1, 6-5, 7-1
- LARGEINT constant 7-3
- las graphical parameter 3-31, 8-21
- LASTENT attribute 7-13
- laxisz graphical subroutine 8-22
- lcopy algorithm 7-14
- leaps function 3-18
- LEFT side of plot 8-6
- len function 2-6, 2-14

LENGTH attribute 6-5, 6-10, 7-12
 less than operator < 2-1
 or equal operator <= 2-1
 letters, upper and lower case 1-2
 lexical analyzer 5-1
 lfill algorithm 7-14
 LGL 3-7
 LIKE specifier 6-4
 linear algebra algorithms 7-18
 algebra, numerical 3-22
 equations 3-22
 model 2-12, 3-18
 lines function 2-19, 3-36
 linesp graphical subroutine 8-25
 linesz graphical subroutine 8-5
 LINEWIDTH constant 7-6
 lintfz graphical subroutine 8-25
 list function 2-7, 2-20, 3-13
 list, description 6-2
 list= argument to rm function 3-13
 literal, ?(macro 4-5
 log function 2-15
 graphical parameter 2-17
 log-linear model 3-19
 log-normal distribution 2-12
 log10 function 2-15
 logarithmic axis 8-21
 plot 2-17, 3-36, 3-37, 6-26
 logaxz graphical subroutine 8-21
 logical operator 2-1
 value 2-5
 logistic distribution 2-12
 loglin function 3-19
 LOGPLOT statement 6-26
 LOOP macro 4-12
 argument, %\$ 4-8
 loop, for 3-5
 LOOP, macro 4-7
 lower case letters 1-2
 lowess function 3-29
 lty graphical parameter 2-19, 3-31, 8-8
 lwd graphical parameter 8-8
 m4 macro processor 7-13
 macro 3-1, 6-1
 argument default value 3-2, 4-3
 argument, \$* 4-7, 4-13
 arbitrary number of 4-7
 positional 4-3
 white space in 4-6
 buffer size 4-11
 compound expression, ({ 3-3, 4-4
 dataset 4-1
 definition 3-2

 print 3-2
 prompt, D> 4-2
 N> 4-2
 documentation 3-17, 4-5
 editor 3-2
 expansion 4-1
 extract 3-15
 function 5-2
 IFELSE 4-6
 invocation, ? 3-1, 4-1, 5-1
 literal, ?(4-5
 LOOP 4-7
 named argument 4-7
 processor, m4 7-13
 MACRO statement 4-2
 macro substitution 4-1, 4-10
 time 3-3, 4-2
 macro, act like function 4-4
 col 3-4
 Cp 3-19
 csweep 3-4
 DEFINE 4-11, 4-12
 discr 3-21
 extract 5-10
 FATAL 4-9
 IFDEF 4-12
 IFELSE 4-12
 LOOP 4-12
 MESSAGE 4-9
 pairs 3-27
 profile 3-8
 prompt 3-17
 PROMPT 4-9, 4-11
 qqplot 3-20, 3-26
 row 3-4
 rsweep 3-4
 same name as function 4-2
 ssweep 3-5
 SUBSTR 4-11, 4-13
 SYS 4-9, 4-13
 value of 3-3, 4-4
 mai graphical parameter 8-17
 mail function 2-4
 main graphical parameter 3-33
 MAKE utility 6-7, 7-2, 8-30
 map of United States 3-24
 mar graphical parameter 3-34, 8-17
 margin coordinate-system 8-2, 8-18
 margins, graphical 3-33, 8-2, 8-16
 outer 3-34, 3-37, 8-2
 match algorithm 6-11
 function 2-14
 match, argument partial 6-17
 partial 6-11

match,

matlines function 3-37
matplot function 2-17, 3-33
matpoints function 3-37
matrix 2-8
 decompositions 3-22
 function 2-8
 plot 2-17
 read 2-8, 2-19
 subset 2-8, 2-11
matrix, diagonal of 2-9
 distance 3-21
 identity 2-9
 transpose of 2-9
max function 2-16
mdscale function 3-22
mean function 2-11
median function 2-11
medit function 3-2, 4-3
MESSAGE macro 4-9
 statement 7-2
message, error 6-3, 7-2
mex graphical parameter 8-17, 8-18
mfcol graphical parameter 3-34
mfcolz graphical subroutine 8-17
mfg graphical parameter 8-17
mfrow graphical parameter 3-34
mfrowz graphical subroutine 8-17
mgp graphical parameter 8-21
min function 2-16
missing argument 5-6
MISSING argument 6-3
 attribute 6-5, 6-17
missing data (see NA) 2-5
MODE attribute 6-10, 7-12
mode function 2-6
 of argument 6-2
 specifier 6-2
mode, alphanumeric 8-28
 built-in 6-16
 CHAR 7-3
 graph 8-28
 INT 7-3
 REAL 7-3
MODECALC specifier 6-15
model, linear 2-12, 3-18
 log-linear 3-19
modes, built-in 3-7
modulo operator %% 2-1
mprint function 3-2, 4-3
mstr function 3-23
mtext function 3-37
mtextz graphical subroutine 8-6, 8-18
multidimensional scaling 3-22
multiplication operator * 2-1

Index

match,

multivariate analysis 3-20
 plotting 3-27
N> macro definition prompt 4-2
NA 2-5, 7-17
 function 2-5
 in data structure 6-12
 graphical functions 3-29
 interface language, operating on 6-12
NA, reading 2-7
 set 6-12
 test for 6-12
name 2-2, 5-1
NAME attribute 7-12
name suffix, file 6-6, 7-1
name, reserved 5-4
named argument 2-2, 6-18
 = 2-3
 argument, macro 4-7
NAOK specifier 6-12
narang algorithm 7-17, 8-5
NBPC constant 7-3
NCOL attribute 6-5, 6-10
ncomp function 3-8, 3-23
NCPW constant 7-3
NDIGITS constant 7-3
NEWDOC utility 3-17, 6-8
newline 3-38
newline, 3-38
next syntax 5-9
NEXTARG statement 6-19
nonparametric statistics 2-14
normal distribution 2-12
 q-q plot 2-18
not available (see NA) 2-5
 equal operator != 2-1
 operator ! 2-1
nper function 2-10
nper, time-series parameter 2-10
nplotz graphical subroutine 8-25
npntsz graphical subroutine 8-25
NROW attribute 6-5, 6-10
null value 3-16
 in iteration 3-16
NULL, built-in 3-16
number, integer 2-5
 real 2-5
numeric labels, axis 8-6
numerical linear algebra 3-22
 values for axis labels 3-33
oma graphical parameter 8-17
omi graphical parameter 8-17
omo graphical parameter 8-19
opening, file 7-10

operating on NA in interface language
6-12

system 4-9

escape ! 2-4

system, escape to 3-10

operator 2-1, 5-7

!, not 2-1

!=, not equal 2-1

\$, component selection 2-11

%%, modulo 2-1

remainder 2-1

%, special 5-1

%, integer division 2-1

%m, coerce 3-7

&, and 2-1

**, exponentiation 2-1

*, multiplication 2-1

+, addition 2-1

-, subtraction 2-1

/, division 2-1

:, sequence 2-1

<, less than 2-1

<-, assignment 2-1, 2-3

<=, less than or equal 2-1

=, equal 2-1, 2-3

>, greater than 2-1

>=, greater than or equal 2-1

precedence 3-7

[, subset 2-2

^, exponentiation 2-1

|, or 2-1

operator, %* 3-22

%c 3-22

%o 3-22

arithmetic 2-1

boolean 2-1

logical 2-1

optional argument 2-2

OPTIONAL argument 6-3

optional argument 6-3

options function 3-1, 4-11, 5-8

or operator | 2-1

order function 2-14

orderf algorithm 7-16

orderi algorithm 7-16

ordering permutation 2-14

orderr algorithm 7-16

orderv algorithm 7-16

otextz graphical subroutine 8-19

outer margins, graphical 3-34, 3-37, 8-2,
8-16

pairs function 3-27

macro 3-27

PAR argument 6-22, 6-24

par function 3-34

PAR graphical parameter 8-11

PAR, graphical parameters 6-22, 6-24

parameters for distributions 2-13

PAR, graphical 6-22, 6-24

parameters, axis 8-22

changing graphical 8-3

graphical 3-31, 8-3

pardump function 3-33, 8-24

parser 5-1

partial match 6-11

match, argument 6-17

pbeta function 3-20

pcauchy function 3-20

pch graphical parameter 2-19, 3-31

pchisq function 3-20

permutation, ordering 2-14

persp function 3-26

graphical subroutine 8-25

perspective plot 3-26

pf function 3-20

graphical utility 8-25

pgamma function 3-20

photo function 3-38

PI constant 7-3

pictur graphical subroutine 8-25

pie chart plot 3-25

function 3-25

piez graphical subroutine 8-25

pin graphical parameter 8-19

plclust function 3-29

plnorm function 3-20

plogis function 3-20

plot data structure 6-25

function 2-17, 2-21

region, graphical 3-33

type, graphical 8-17

plot, bar 3-25

box 3-23

cluster tree 3-29

contour 3-26

Cp 3-19

dendrogram 3-29

empirical q-q 2-18

face 3-27

graphical 8-2, 8-16

histogram 2-18

logarithmic 2-17, 3-36, 3-37, 6-26

matrix 2-17

normal q-q 2-18

perspective 3-26

pie chart 3-25

polygon 3-27

probability 2-18

- q-q 2-18
- quantile-quantile 2-18
- scatter 2-17, 3-27
- star 3-27
- stem-and-leaf 2-18
- time-series 2-17
- two-way fit 3-25
- plot1 graphical subroutine 8-25
- plot= argument 3-29
- PLOTARGS argument 6-25
 - statement 8-11
- plotfit function 3-25
- plotter, Hewlett-Packard 2-16
 - Tektronix 2-16
- plotting dot 2-19
 - symbols 2-19
- plotting, multivariate 3-27
- pltusa graphical subroutine 8-25
- pmax function 2-16
- pmin function 2-16
- pnorm function 3-20
- pnttfz graphical subroutine 8-25
- pointer 7-11
 - data 6-11
- POINTER statement 7-3
- points function 2-19, 3-36
- pointz graphical subroutine 8-5
- polygon plot 3-27
- pos= argument (database position) 2-8
- positional macro argument 4-3
- ppoints function 3-38
- prcomp function 3-20
- PRECISION constant 7-3
- prefix function 3-12
- prefix, effect on list function 3-13
- pretty function 3-33
 - numbers for axis labels 8-21
- principal components 3-20
- print algorithm facility 7-4
 - function 2-9, 2-20
- PRINT statement 7-4
- print, automatic 2-1, 2-2, 3-7, 5-8
 - macro definition 3-2
 - table 3-17, 3-19
- PRINTDOC utility 3-18, 6-8
- printer function 2-16, 2-21
- printer, show function 2-19
- prism function 3-38
- probability algorithms 7-17
 - functions 2-13, 3-20
 - plot 2-18
 - plot, gamma 3-26
 - plots of general distributions 3-26
- prod function 2-16
- profile macro 3-8
- PROGRAM utility 7-2, 8-10
- prompt macro 3-17
- PROMPT macro 4-9, 4-11
 - utility 6-7
- prompt, + continuation 2-3
 - > 2-1
 - D> macro definition 4-2
 - N> macro definition 4-2
- pseudorandom number generator 2-12
- pstab function 3-20
- pt function 3-20
- ptitle graphical subroutine 8-25
- pty graphical parameter 3-34, 8-17
- punif function 3-20
- PUTSEED statement 7-18
- pwd command 6-7
- Q format item 7-5
- q function 2-23
- q-q plot 2-18
- q-r decomposition 7-18
- qbeta function 3-20
- qcauchy function 3-20
- qchisq function 3-20
- qf function 3-20
- qgamma function 3-20
- qlnorm function 3-20
- qlogis function 3-20
- qnorm function 3-20
- qqgamma function 3-26
- qqnorm function 2-18, 3-29
- qqplot function 2-18, 3-29
 - macro 3-20, 3-26
- qstab function 3-20
- qt function 3-20
- qualitative data 3-16
- quantile algorithms 7-17
 - functions 2-13, 3-20
- quantile-quantile plot 2-18
- QUERY statement 8-7, 8-23
- qunif function 3-20
- R format item 7-5, 7-9
- random number algorithms 7-17
 - generator 2-12
- Random.seed dataset 7-17
- range function 2-16, 3-36
- range algorithm 7-16, 8-5
- rangev algorithm 7-16, 8-5
- rank function 2-14
 - test, two-sample 2-14
- raster coordinate system 8-26
- RATFOR language 6-1, 6-5, 7-1
- rbeta function 2-12

- rbind function 2-9
- rbiwt function 2-12
- rcauchy function 2-12
- rchisq function 2-12
- rcopy algorithm 7-14
- rdpen function 3-29, 3-30
 - graphical utility 8-26
- rdtfmt algorithm 7-7
- read algorithm facility 7-8
 - function 2-6, 5-10
 - self-describing data 3-14
- READ statement 7-9
- read, character string 2-20
 - strings 2-6
 - from file 2-6, 3-15
 - terminal 2-6
 - matrix 2-8, 2-19
 - NA 2-7
- REAL 3-7
 - mode 7-3
- real number 2-5
- reg function 3-19
- regprt function 3-19
- regress function 2-11, 2-12, 2-22
- regression, robust 2-12, 3-18
 - stepwise 3-18
 - subsets 3-18
 - time-series for 2-10
- regsum function 3-19
- remainder operator %% 2-1
- remove, dataset from database 2-7
- removing intermediate dataset 4-4
- rep function 2-13
- repeat syntax 5-9
- replace function 2-6
- replot function 3-40
- replotting deferred graphics 3-39
- reproducing samples 2-13
- reserved name 5-4
- restore function 3-14, 5-10
- RETURN statement 6-6, 6-22
- rev function 2-14
- rf function 2-12
- rfill algorithm 7-14
- rgamma function 2-12
- RIGHT side of plot 8-6
- rlnorm function 2-12
- rlogis function 2-12
- rm function 2-7, 3-13
 - function, value= argument to 4-4
- rnorm function 2-12
- robust regression 2-12, 3-18
 - smoothing 2-15
- round function 2-16
- row function 2-9
 - macro 3-4
- rplotz graphical subroutine 8-25
- rpntsz graphical subroutine 8-25
- rreg function 3-18
- rstab function 2-12
- rsweep macro 3-4
- rt function 2-12
- runif function 2-12
- S environment 7-1
 - format item 7-9
- samples, reproducing 2-13
- sapply function 3-4
- sattac algorithm 7-11
- save database 2-7
 - function 2-7, 5-7
- saxisz graphical subroutine 8-6, 8-22
- scale function 3-20
- scaling, correlation 3-20
 - multidimensional 3-22
- scatter plot 2-17, 3-27
- sclose algorithm 7-11
- scratch space allocation 6-4
- sdetac algorithm 7-11
- search function 3-14, 6-7
 - list 3-13
 - list, database 2-7
- segments function 3-37
- segmtz graphical subroutine 8-5
- select function 3-10
- select, component 6-13
- self-describing data structure 2-5
 - data, read/write 3-14
- semantics 5-5
- seq function 2-13
- sequence operator : 2-1
- set NA 6-12
- SETUP statement 6-25, 8-11
- shadez graphical subroutine 8-25
- shared database 2-7
- SHARP format item 7-8
- show function 2-19, 2-21
- sin function 2-15
- singular-value decomposition 7-18
- sink function 3-1
- SMALL constant 7-3
- smatrix function 3-28
- smooth function 2-15
- smoothing, robust 2-15
- solve function 3-22
- sopen algorithm 7-11
- sort algorithm 7-15
 - function 2-14
- sort, descending order 2-14

- several fields 2-14
- sortfr algorithm 7-15
- sorti algorithm 7-15
- sortv algorithm 7-16
- source function 3-1
- SP format item 7-7
- special operator % 5-1
- specifier, ANY 6-14
 - FROM 6-13
 - LIKE 6-4
 - mode 6-2
 - MODECALC 6-15
 - NAOK 6-12
 - STR 6-13
 - TSTRING 7-15
 - type 6-2
- SPECIFY statement 8-7, 8-23
- split function 3-5, 3-22
- splotz graphical subroutine 8-25
- sqr function 2-15
- srt graphical parameter 3-32, 3-36, 8-7
- ssweep macro 3-5
- stable distribution 2-12
- stack algorithm facility 7-11
- stand-alone device driver 8-25
- star plot 3-27
- stare function 3-38
- stars function 3-27
- starsymb function 3-28
- start function 2-10
- start, time-series parameter 2-10
- state.center, dataset 3-24
- statement, ABORT 7-2, 7-5
 - ALLARG 6-19
 - ARG 6-18
 - ARGSTR 6-21
 - AXIS 8-5, 8-20
 - CASE 6-15
 - CHAIN 6-23, 8-12
 - CHARACTER 7-3
 - CLEAR 7-7
 - COERCE 6-16
 - CTABLE 6-11
 - DEBUG 6-8, 7-3
 - DECODE 7-9
 - DEVICE_DRIVER 8-30
 - ENCODE 7-6
 - ENDARGS 6-18
 - EPRINT 7-5
 - FATAL 6-4, 7-2
 - FPRINT 7-5
 - FUNCTION 6-2
 - GETLINE 7-10
 - GETSEED 7-17

- GETSTRING 7-10
- INCLUDE 7-4
- INSERT 6-23
- LOGPLOT 6-26
- MACRO 4-2
- MESSAGE 7-2
- NEXTARG 6-19
- PLOTARGS 8-11
- POINTER 7-3
- PRINT 7-4
- PUTSEED 7-18
- QUERY 8-7, 8-23
- READ 7-9
- RETURN 6-6, 6-22
- SETUP 6-25, 8-11
- SPECIFY 8-7, 8-23
- STATIC 6-5
- STRUCTURE 6-4
- SWITCH MODE 6-15
- TREEWALK 7-13
- WARNING 7-2
- STATIC statement 6-5
- statistical distribution 2-12
- statistics, distribution-free 2-14
 - nonparametric 2-14
- stdaxz graphical subroutine 8-21
- stem function 2-18, 2-20, 2-23
- stem-and-leaf plot 2-18
- stepwise regression 3-18
- storage allocation, dynamic 7-11
- STR specifier 6-13
- streq algorithm 7-15
- STRING attribute 6-11
- string data, character 6-10, 7-6
- string, character 5-1
- struct algorithm facility 7-12
- STRUCTURE statement 6-4
- structure, argument 5-5
 - list from 6-20
 - data 2-8, 5-9, 7-12
 - depth-first scan of data 7-13
 - dynamically allocated data 6-4
 - external representation of data 5-10
 - hierarchical 3-23, 6-13
 - internal representation of data 5-10
 - NA in data 6-12
 - vector 3-11
- Student's t distribution 2-12
- style, axis 8-21, 8-22
- sub graphical parameter 3-33
- subset operator [2-2
- subset, assignment to 2-2

- interface language 6-5
 - matrix 2-8, 2-11
 - time-series 2-11
- subsets regression 3-18
- substitution, macro 4-1, 4-10
- SUBSTR macro 4-11, 4-13
- subtraction operator - 2-1
- subtree function 3-21
- suffix, file name 6-6, 7-1
- sum function 2-16
- svd function 3-22
- sweep function 3-4
- SWITCH MODE statement 6-15
- symbols, plotting 2-19
- syntax error 2-3
 - rule 5-1
- syntax, break 5-9
 - else 3-6
 - for 3-5, 5-9
 - if 3-6
 - next 5-9
 - repeat 5-9
 - while 5-9
- sys function 3-10
- SYS macro 4-9, 4-13
- system database 2-7
- T format item 7-7
- t function 2-9
- table function 3-17, 3-19
 - print 3-17, 3-19
- taxisz graphical subroutine 8-22
- tck graphical parameter 3-32, 8-21
- tek10 function 2-16
- tek12 function 2-16
- tek14 function 2-16
- tek14q function 2-16
- tek46f function 2-16
- tek46h function 2-16
- tek46v function 2-16
- Tektronix plotter 2-16
 - terminal 2-16
- TEND attribute 6-5, 6-10
- terminal, graphical 2-16
 - Tektronix 2-16
- test for NA 6-12
- TEXT attribute 6-11
- text editor 2-4, 3-1, 3-15, 3-17, 4-2, 4-3
 - function 2-19, 3-36
- textz graphical subroutine 8-6
- tick marks, axis 8-6
- timaxz graphical subroutine 8-21
- time axis 8-21
 - function 2-10
- time-series 2-10
 - arithmetic 2-10
 - for regression 2-10
 - plot 2-17
 - subset 2-11
- title function 2-19
- title, graphical 8-6
- titlez graphical subroutine 8-25
- TM format item 7-7
- TNPER attribute 6-5, 6-10
- token 5-1
- TOP side of plot 8-6
- tprint function 3-17, 3-19
- transpose of matrix 2-9
- tree algorithm facility 7-13
- TREEWALK statement 7-13
- triangular systems of equations 3-22
- TRUE 2-1, 2-5
- trunc function 2-16
- ts function 2-10
- tslines function 3-37
- tsmatrix function 2-10
- Tsp component 3-12
- tsplot function 2-17, 3-33
- tspltz graphical subroutine 8-25
- tspoints function 3-37
- TSTART attribute 6-5, 6-10
- TSTRING specifier 7-15
- tsvecz graphical subroutine 8-25
- two-sample rank test 2-14
- two-way fit plot 3-25
- twoway function 2-12
- type graphical parameter 2-17, 3-33
 - of argument 6-2
 - specifier 6-2
- type, axis 8-22
- uin graphical parameter 8-20
- uniform distribution 2-12
- unique function 2-15
- United States, map of 3-24
- upper case letters 1-2
- usa function 3-24
- usabdy graphical subroutine 8-25
- USED argument 6-21
- user coordinate system 3-33, 8-2, 8-20, 8-22
 - coordinates, equal size 3-36
- usr graphical parameter 3-35, 8-20, 8-22
- utility, BATCH 3-9
 - BPLOT 3-39
 - CHAPTER 6-6
 - EDITDOC 3-17, 6-8
 - extract 3-15, 5-10, 7-9
 - FUNCTION 6-6, 6-24, 7-1, 7-17, 8-10, 8-29

- GRAPH 8-25
- MAKE 6-7, 7-2, 8-30
- NEWDOC 3-17, 6-8
- pf graphical 8-25
- PRINTDOC 3-18, 6-8
- PROGRAM 7-2, 8-10
- PROMPT 6-7
- rdpen graphical 8-26
- VALUE attribute 6-17, 7-12
- value of compound expression 3-7, 5-8
 - conditional expression 3-6
 - macro 3-3, 4-4
- value= argument to rm function 4-4
- var function 2-12
- variable, BUFFER 7-6
 - BUFLen 7-6
 - BUFPOS 7-6
 - EOF 7-10
 - FIELD 7-10
 - FIELDPOS 7-10
 - INBUF 7-8
 - INLEN 7-8
 - INPOS 7-8
- variance, var function 2-12
- vector 1-1, 2-5
 - structure 3-11
- vu function 3-26
- WARNING statement 7-2
- while syntax 5-9
- white space in macro argument 4-6
- window function 2-11
- working database 2-7
- write function 3-14, 5-10
 - self-describing data 3-14
- xaxp graphical parameter 8-22
- xaxs graphical parameter 3-32, 8-22
- xaxt graphical parameter 3-32, 8-22
- xlab graphical parameter 3-33
- xlim graphical parameter 3-33
- xpd graphical parameter 8-19
- yaxp graphical parameter 8-22
- yaxs graphical parameter 3-32, 8-22
- yaxt graphical parameter 3-32, 8-22
- ylab graphical parameter 3-33
- ylim graphical parameter 3-33
- zejecz graphical subroutine 8-28
- zinitz graphical subroutine 8-15
- zlinez graphical subroutine 8-28
- zoutrz graphical subroutine 8-29
- zoutwz graphical subroutine 8-29
- zparmz graphical subroutine 8-27
- zptchz graphical subroutine 8-28
- zquxyz graphical subroutine 8-29
- zseekz graphical subroutine 8-28
- zwrapz graphical subroutine 8-28

Suggestions

We would like to receive your comments about S. Let us know your reactions to the documentation, functions, and overall design of the system. We would also appreciate bug reports, suggestions for future developments, information about new functions you implement, etc.

Although it is impossible to provide personal answers to inquiries about S, your comments and questions will be used to improve further releases of S.

Send responses to:

Bell Laboratories
Computing Information Service
600 Mountain Avenue
Murray Hill, New Jersey 07974